
# Software Diversification for WebAssembly

**JAVIER CABRERA-ARTEAGA**

Doctoral Thesis in Computer Science
Supervised by
Benoit Baudry and Martin Monperrus

Stockholm, Sweden, 2023

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
Division of Software and Computer Systems
SE-10044 Stockholm
Sweden

TRITA-EECS-AVL-2020:4
ISBN 100-

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av Teknologie doktorexamen i elektroteknik i .



Tryck: Universitetsservice US AB

**Abstract**

**Keywords:** Lorem, Ipsum, Dolor, Sit, Amet

**Sammanfattning**

# LIST OF PAPERS

1. ***WebAssembly Diversification for Malware Evasion***
   **Javier Cabrera-Arteaga**,Tim Toady, Martin Monperrus, Benoit Baudry
   *Computers & Security, Volume 131, 2023, 17 pages*
   https://www.sciencedirect.com/science/article/pii/S0167404823002067

2. ***Wasm-mutate: Fast and Effective Binary Diversification for WebAssembly***
   **Javier Cabrera-Arteaga**, Nicholas Fitzgerald, Martin Monperrus, Benoit Baudry
   *Under review, 17 pages*
   https://arxiv.org/pdf/2309.07638.pdf

3. ***Multi-Variant Execution at the Edge***
   **Javier Cabrera-Arteaga**, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
   *Moving Target Defense (MTD 2022), 12 pages*
   https://dl.acm.org/doi/abs/10.1145/3560828.3564007

4. ***CROW: Code Diversification for WebAssembly***
   **Javier Cabrera-Arteaga**, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
   *Measurements, Attacks, and Defenses for the Web (MADWeb 2021), 12 pages*
   https://doi.org/10.14722/madweb.2021.23004

5. ***Superoptimization of WebAssembly Bytecode***
   **Javier Cabrera-Arteaga**, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
   *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs, 4 pages*
   https://doi.org/10.1145/3397537.3397567

6. ***Scalable Comparison of JavaScript V8 Bytecode Traces***
   **Javier Cabrera-Arteaga**, Martin Monperrus, Benoit Baudry
   *11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019), 10 pages*
   https://doi.org/10.1145/3358504.3361228

# ACKNOWLEDGEMENT

# Contents

# Part I

# Thesis

# 1 INTRODUCTION

*Jealous stepmother and sisters; magical aid by a beast; a marriage won by gifts magically provided; a bird revealing a secret; a recognition by aid of a ring; or show; or what not; a dénouement of punishment; a happy marriage - all those things, which in sequence, make up Cinderella, may and do occur in an incalculable number of other combinations.*

— MR. Cox ***1893***, *Cinderella: Three hundred and forty-five variants [**?** ]*

The first web browser, Nexus, made its appearance in 1990 [**?** ]. At its inception, web browsing consisted solely of retrieving and displaying small, static text pages. In other words, users could only read page content without any interactive components. However, the escalating computing power of devices, the proliferation of the internet, the valuation of internet-based companies and the demand for more engaging user experiences gave rise the concept of executing code in conjunction with web pages. In 1995, the Netscape browser revolutionized this concept by introducing JavaScript [**?** ], a programming language that allowed code execution on the client-side. Interactive web content immediately highlighted benefits: unlike classical native software, web applications do not require installation, are always up-to-date, and are accessible from any device with a web browser. Significantly, since the advent of Netscape, all browsers offer JavaScript support. In the present day, the majority of web pages incorporate not only HTML but also JavaScript code, which is executed on client computers. Over the past several decades, web browsers have transformed into JavaScript virtual machines. They have evolved into intricate systems capable of running comprehensive applications, such as video and audio players, animation creators, and PDF document renderers.

JavaScript is presently the most widely utilized scripting language in all contemporary web browsers [**?** ]. However, it is not without limitations due to the inherent characteristics of the language. For instance, each JavaScript engine needs the parsing and recompiling of the JavaScript code, resulting in substantial overhead. Just parsing and compiling JavaScript code consumes the majority of the load times of websites [**?** ]. In addition to performance limitations,

[0] Compilation probe time 2023/11/07 12:55:02

JavaScript also has security concerns [**?** ]. A notable example of this is the lack of memory isolation in JavaScript, which allows extraction of information from other processes [**?** ]. These issues led the Web Consortium (W3C) to standardize a bytecode for the web environment in 2015, which is the WebAssembly (Wasm) bytecode. Hence, WebAssembly became the fourth official language for the web.

WebAssembly is designed with a focus on speed, portability, self-containment, and security [**?** ]. It enables the ahead-of-time compilation of all programs from source languages such as C/C++ and Rust. Third-party compilers produce WebAssembly binaries, as is the case of LLVM. WebAssembly bytecode format abstracts its Instruction Set Architecture, making it akin to machine code instructions but independent of CPU architectures [**?** ]. Resembling machine code, WebAssembly is already optimized and consists of consecutive binary sections. The contiguous array organization of a WebAssembly binary enables efficient processing, allowing compilers to speed up compilation through parallel parsing. Wasm binaries not only validate and compile rapidly but are also quick to transmit over a network due to their small sizes.

WebAssembly's versatility extends beyond web browsers to backend scenarios. Studies have highlighted the benefits of using WebAssembly as an intermediate layer, including improved startup times and enhanced memory usage [**?** **?** ]. The Bytecode Alliance consequently proposed the WebAssembly System Interface (WASI) in 2019 [**?** **?** ]. WASI standardized the execution of WebAssembly utilizing a POSIX system interface protocol, thus enabling WebAssembly's direct execution in the operating system.

## 1.1 WebAssembly security

WebAssembly is praised for its security, especially for its design that prevents programs from accessing data beyond their own memory. However, there has been less focus on potential vulnerabilities and attacks within WebAssembly's own memory [**?** ].

Remarkably, WebAssembly binaries can have inherent vulnerabilities due to source code flaws. For example, the absence of stack-smashing protections like stack canaries in code compiled to WebAssembly could lead to undetected overflows in WebAssembly, causing crashes in standalone deployments [**?** ] .

Moreover, significant risks exist from side-channel attacks on WebAssembly. Rokicki et al. revealed the risk for port contention side-channel attacks on WebAssembly binaries in browsers [**?** ]. In standalone deployments, Genkin et al. demonstrated the potential for data extraction via cache timing-side channels in WebAssembly [**?** ]. Similarly, Maisuradze and Rossow showed speculative execution attacks on WebAssembly binaries [**?** ].

## 1.2 Software Monoculture

Web browsers and JavaScript have evolved significantly in the past thirty years, leading to numerous implementations. Yet, only Firefox, Chrome, Safari, and Edge are commonly used on devices. This situation reflects a software monoculture problem wherein a single flaw could impact multiple applications. The concept of monoculture is borrowed from biology and symbolizes an ecosystem at risk of extinction due to shared vulnerabilities and lack of diversity. Currently, web pages including WebAssembly binaries are centrally served from main datacenters. Thus, this monoculture issue is also applicable to the WebAssembly code served to web browsers. Therefore, sharing Wasm code through web browsers could also share its vulnerabilities.

The software monoculture problem exacerbates when considering the edge-cloud computing platforms and their adoption of WebAssembly to provide services. Specifically, in addition to browser clients, thousands of edge devices running WebAssembly as backend services could be affected by shared vulnerabilities. This scenario suggests that if one node in an edge network is vulnerable, all the others would be vulnerable in the exact same way since the same binary is replicated on each node. In other words, the same attacker payload could compromise all edge nodes simultaneously, meaning that a single distributed Wasm binary could trigger a worldwide attack.

## 1.3 WebAssembly malware evasion

WebAssembly is often used in browsers for computation-intensive activities, including gaming and image processing, but it has also been exploited by malicious actors for cryptojacking [**?** ]. The popularity of WebAssembly for cryptojacking stems from its ability to execute a high volume of hash functions. Since WebAssembly outperforms JavaScript in speed, it is the natural option for cryptojacking. Besides, WebAssembly code's poor readability makes it a convenient tool for obfuscating harmful code. Cryptojacking via WebAssembly often involves a malicious JavaScript+WebAssembly payload that secretly executes on the victim's browser and generates passive income [**?** ]. Because it is hard to detect and remove, cryptojacking can execute on a victim's computer, continuously using resources and generating income for the attacker.

Several techniques employ static analysis, dynamic analysis and even state-of-the-art machine learning methods to detect WebAssembly crytomalware [**? ? ? ? ? ?** ]. Obfuscation studies have revealed weaknesses in several of these methods, indicating a largely unexplored threat to malware detection accuracy in WebAssembly. Yet, the majority of these studies do not consider the existence of obfuscation tools.

## 1.4 Problems statements

According to the discussion above, we identify three key problems to be addressed.

**Ps1 WebAssembly security**: WebAssembly ecosystem and binaries are vulnerable to attacks, specially side-channel threats. Existing WebAssembly research mostly reacts to existing vulnerabilities, leaving the potential for unidentified attacks. Besides, current defenses are limited to specific attacks or require the alteration of runtimes.

**Ps2 Software monoculture**: Identical WebAssembly binaries are deployed on multiple nodes and browsers. Deployment systems, including web browsers, might be also identical. Such a situation presents a potential threat to the entire ecosystem due to shared vulnerabilities.

**Ps3 WebAssembly malware evasion**: WebAssembly malware is a serious threat. Current implementated defenses are not sufficient to protect against WebAssembly malware, mostly because current defenses ignore malware obfuscation.

## 1.5 Software Diversification

This dissertation introduces tools, strategies, and methodologies designed to address the previously enunciated problem statements via Software Diversification. Software Diversification is a security-focused process that involves identifying, developing, and deploying program variants of a given original program [**?** ]. Pioneers in this field, Cohen et al. [**?** ] and Forrest et al. [**?** ], proposed enhancing software diversity through code transformations. Their proposal suggested creating program variants while maintaining their functionalities to mitigate potential vulnerabilities.

Software diversification, as demonstrated in previous studies, can effectively remove vulnerabilities. For instance, Eichin et al. [**?** ] presented seminal work in 1989, illustrating the practical benefits of diversification. Specifically, the diversification limited the Morris Worm's exploitation only to a few machines. However, despite extensive research, the use of software diversification in WebAssembly remains largely unexplored.

Software diversification could bolster WebAssembly analysis tools by incorporating diversified program variants, thereby hardening the attackers' task of exploiting vulnerabilities. Generated variants, created proactively for security, could emulate a broad spectrum of real-world conditions, subsequently improving the accuracy of WebAssembly analysis tools, including WebAssembly malware detectors. Moreover, current solutions to mitigate side-channel attacks on WebAssembly binaries either target specific attacks or need the modification of runtimes. Thus, by generating diversified variants independent of the

| Contribution | Research papers | | | |
|---|---|---|---|---|
| | P1 | P2 | P3 | P4 |
| C1 Experimental contribution | ✓ | ✓ | ✓ | ✓ |
| C2 Theoretical contribution | ✓ | | ✓ | |
| C3 Diversity generation | ✓ | ✓ | ✓ | ✓ |
| C4 Defensive diversification | ✓ | ✓ | ✓ | |
| C5 Offensive diversification | | | | ✓ |

***Table 1.1:*** *Mapping between contributions and research papers .*

platform, software diversification could help address potential vulnerabilities in WebAssembly binaries. In the context of software diversification, we present the following, non-necesarilly orthogonal, contributions.

**C1 Experimental contribution:** For each proposed technique we provide an artifact implementation and conduct experiments to assess its capabilities. The artifacts are publicly available. The protocols and results of assessing the artifacts provide guidance for future research.

**C2 Theoretical contribution:** We propose a theoretical foundation in order to generate and improve Software Diversification for WebAssembly. We provide a formal definition of WebAssembly program variants and their diversity. We also provide a formal definition of WebAssembly program diversity generation.

**C3 Diversity generation:** We generate WebAssembly program variants. The variants are functionally equivalent to the original program, yet behaviorally diverse.

**C4 Defensive Diversification:** We assess how generated WebAssembly program variants could be used for defensive purposes. We provide empirical insights about the practical usage of the generated variants in preventing attacks.

**C5 Offensive Diversification:** We evaluate the potential for using generated WebAssembly program variants for offensive purposes. Our research includes experiments where we test the resilience of WebAssembly analysis tools against these generated variants. Furthermore, we offer insights into which types of program variants practitioners should prioritize to improve WebAssembly analysis tools.

## 1.6 Summary of research papers

This compilation thesis comprises the following research papers. In Table 1.1 we map the contributions to our research papers.

**P1**: **CROW: Code randomization for WebAssembly bytecode.**
**Javier Cabrera-Arteaga**, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
*Measurements, Attacks, and Defenses for the Web (MADWeb 2021), 12 pages*
`https://doi.org/10.14722/madweb.2021.23004`

**Summary:** In this paper, we introduce the first entirely automated workflow for diversifying WebAssembly binaries. We present CROW, an open-source tool that implements software diversification through enumerative synthesis. We assess the capabilities of CROW and examine its application on real-world, security-sensitive programs. In general, CROW can create many statically diverse variants. Furthermore, we illustrate that the generated variants exhibit different behaviors at runtime.

**P2**: **Multivariant execution at the Edge.**
**Javier Cabrera-Arteaga**, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
*Moving Target Defense (MTD 2022), 12 pages*
`https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

**Summary:** In this paper, we synthesize functionally equivalent variants of deployed edge services. Service variants are encapsulated into a single multivariant WebAssembly binary. A random variant is selected and executed each time a function is invoked. Execution of multivariant binaries occurs on the global edge platform provided by Fastly, as part of a research collaboration. We demonstrate that multivariant binaries present a diverse range of execution traces throughout the entire edge platform, distributed worldwide, effectively creating a moving target defense.

**P3**: **Wasm-mutate: Fast and efficient software diversification for WebAssembly.**
**Javier Cabrera-Arteaga**, Nicholas Fitzgerald, Martin Monperrus, Benoit Baudry
*Under review, 17 pages*
`https://arxiv.org/pdf/2309.07638.pdf`

**Summary:** This paper introduces WASM-MUTATE, a compiler-agnostic WebAssembly diversification engine. The engine is designed to swiftly

generate functionally equivalent yet behaviorally diverse WebAssembly variants by randomly traversing e-graphs. We show that WASM-MUTATE can generate tens of thousands of unique WebAssembly variants in minutes. Importantly, WASM-MUTATE can safeguard WebAssembly binaries from timing side-channel attacks, such as Spectre.

**P4**: **WebAssembly Diversification for Malware evasion.**
**Javier Cabrera-Arteaga**, Tim Toady, Martin Monperrus, Benoit Baudry
*Computers & Security, Volume 131, 2023, 17 pages*

**Summary:** WebAssembly, while enhancing rich applications in browsers, also proves efficient in developing cryptojacking malware. Protective measures against cryptomalware have not factored in the potential use of evasion techniques by attackers. This paper delves into the potential of automatic binary diversification in aiming WebAssembly cryptojacking detectors' evasion. We provide proof that our diversification tools can generate variants of WebAssembly cryptojacking that successfully evade VirusTotal and MINOS. We further demonstrate that these generated variants introduce minimal performance overhead, thus verifying binary diversification as an effective evasion technique.

## Thesis layout

This dissertation comprises two parts as a compilation thesis. Part one summarises the research papers included within, which is partially rooted in the author's licentiate thesis [**?** ]. Chapter 2 offers a background on WebAssembly and the latest advancements in Software Diversification. Chapter 3 delves into our technical contributions. Chapter 4 exhibits two use cases applying our technical contributions. Chapter 5 concludes the thesis and outlines future research directions. The second part of this thesis incorporates all the papers discussed in part one.

# 2
# BACKGROUND AND STATE OF THE ART

*You must have a map, no matter how rough. Otherwise you wander all over the place.*

— J.R.R. Tolkien

This chapter discusses the state-of-the-art in the areas of WebAssembly and Software Diversification. In Section 2.1 we discuss WebAssembly, focusing on its design and security model. Besides, we discuss the current state-of-the-art of WebAssembly research. In Section 2.2 we discuss related works in the area of Software Diversification. Moreover, we delve into the open challenges regarding the diversification of WebAssembly programs.

## 2.1 WebAssembly

The W3C publicly announced the WebAssembly (Wasm) language in 2015 as the fourth scripting language supported in all major web browser vendors. WebAssembly is a binary instruction format for a stack-based virtual machine and was officially consolidated by the work of Haas et al. [**?** ] in 2017 and extended by Rossberg et al. in 2018 [**?** ]. It is designed to be fast, portable, self-contained, and secure.

Moreover, WebAssembly has been evolving outside web browsers since its first announcement. Some works demonstrated that using WebAssembly as an intermediate layer is better in terms of startup time and memory usage than containerization and virtualization [**?** **?** ]. Consequently, in 2019, the Bytecode Alliance proposed WebAssembly System Interface (WASI) [**?** ]. WASI pioneered the execution of WebAssembly with a POSIX system interface protocol, making it possible to execute Wasm closer to the underlying operating system. Therefore, it standardizes the adoption of WebAssembly in heterogeneous platforms [**?** ], i.e., IoT and Edge computing [**?** **?** ].

[0] Compilation probe time 2023/11/07 12:55:02

Currently, WebAssembly serves a variety of functions in browsers, ranging from gaming to cryptomining [**?** ]. Other applications include text processing, visualization, media processing, programming language testing, online gambling, bar code and QR code fast reading, hashing, and PDF viewing. On the backend, WebAssembly notably excels in short-running tasks. As such, it is particularly suitable for Function as a Service (FaaS) platforms like Cloudflare and Fastly. The subsequent text in this chapter focuses specifically on WebAssembly version 1.0. However, the tools, techniques, and methodologies discussed are applicable to future WebAssembly versions.

### 2.1.1 From source code to WebAssembly

WebAssembly programs are compiled from source languages like C/C++, Rust, or Go, which means that it can benefit from the optimizations of the source language compiler. The resulting WebAssembly program is like a traditional shared library, containing instruction codes, symbols, and exported functions. A host environment is in charge of complementing the Wasm program, such as providing external functions required for execution within the host engine. For instance, functions for interacting with an HTML page's DOM are imported into the Wasm binary when invoked from JavaScript code in the browser.

```
1  ...
2  // Imported form host
3  extern "C" {
4      fn log(s: &str);
5      fn get_input() -> usize; }
6
7  fn fibo(n: usize) -> i32 {
8      // Iterative fibonacci
9      // Create a vector of size n+1
10     let mut fibo_result = vec![0; n + 1];
11     // Set ith 0 and 1
12     fibo_result[0] = 1;
13     fibo_result[1] = 1;
14     for i in 2..=n {
15         // f[i] = f[i-1] + f[i-2]
16         fibo_result[i] = fibo_result[i - 1] + fibo_result[i - 2];
17     }
18     // Return the last element
19     return fibo_result[n];
20 }
21 // Pub to export the function
22 pub fn main() {
23     // Get the input from the user
24     let ith = get_input();
25     // Calculate the fibonacci
26     let fib = fibo(get_input());
27     // Print the result in the host imported function
28     log(&format!("{}",fib));
29 }
```

***Listing 2.1:*** *Example Rust program which includes, external function usage, a function definition featuring a loop, function calls, imported functions, and memory accesses.*

In Listing 2.1 and Listing 2.2, we present a Rust program alongside its corresponding WebAssembly binary. The Rust program in Listing 2.1 calculates the Fibonacci sequence up to a given number that comes from the host engine. The code in the program encompasses various elements such as vector allocations, external function usage, and a function definition that includes a loop, conditional branching, function calls, and memory accesses. The Wasm code shown in Listing 2.2 is simplified in its textual format, known as WAT[1]. The function prototype in lines 4 and 5 of Listing 2.1 are converted into imported function, as seen in lines 8 and 9 of Listing 2.2. The `fibo` function, spanning lines 7 to 20 in Listing 2.1, is compiled into a Wasm function covering lines 14 to 31 in Listing 2.2. Within this function, the translation of various Rust language constructs into Wasm can be observed. For instance, the `for` loop found in line 14 of Listing 2.1 is mapped to a block structure in lines 17 to 31 of Listing 2.2. The breaking condition of the loop is transformed into a conditional branch, as depicted in line 23 of Listing 2.2. In this scenario, the function yields the final set value in the `local` variable. Note that for optimization purposes, the loop concludes by returning the result value, instead of returning post completion of

[1]The WAT text format is primarily designed for human readability and for low-level manual editing.

the loop.

There exist several compilers that turn source code into WebAssembly binaries. For example, LLVM compiles to WebAssembly as a backend option since its 7.1.0 release in early 2019[2], supporting a diverse set of frontend languages like C/C++, Rust, Go, and AssemblyScript[3]. Significantly, a study by Hilbig [**?** ] reveals that 70% of WebAssembly binaries are generated using LLVM-based compilers. The main advantage of using LLVM is that it provides a modular and state-of-the-art optimization infrastructure for WebAssembly binaries. Recently, the Kotlin Multiplatform framework [4] has incorporated WebAssembly as a compilation target, enabling the compilation of Kotlin code to WebAssembly.

A recent trend in the WebAssembly ecosystem involves porting various programming languages by converting both the language's engine or interpreter and the source code into a WebAssembly program. For example, Javy[5] encapsulates JavaScript code within the QuickJS interpreter, demonstrating that direct source code conversion to WebAssembly isn't always required. If an interpreter for a specific language can be compiled to WebAssembly, it allows for the bundling of both the interpreter and the language into a single, isolated WebAssembly binary. Similarly, Blazor[6] facilitates the execution of .NET Common Intermediate Language (CIL) in WebAssembly binaries for browser-based applications. However, packaging the interpreter and the code in one single standalone WebAssembly binary is still immature and faces challenges. For example, the absence of JIT compilation for the "interpreted" code makes it less suitable for long-running tasks [**?** ]. On the other hand, it proves effective for short-running tasks, particularly those executed in Edge-Cloud computing platforms.

[2] `https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0`
[3] A subset of the TypeScript language
[4] `https://kotlinlang.org/docs/wasm-overview.html`
[5] `https://github.com/bytecodealliance/javy`
[6] `https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor`

```
; WebAssembly magic bytes(\0asm) and version (1.0) ;
(module
  ...
  ; Type section: 0x01 0x00 0x00 0x00 0x13 ... ;
  (type (;type index 0;) (func (param i32 i32)))
  ...
  ; Import section: 0x02  0x00 0x00 0x00 0x57 ... ;
  (import "__wbg__" "__wbg_log" (func (;1;) (type 0)))
  (import "__wbg__" "__wbg_getinput" (func (;2;) (type 8)))
  ...
  ; Custom section: 0x00 0x00 0x00 0x00 0x7E ;
  (@custom "name" "...")
  ...
  (func (;func index 40;) (type 1) (param i32) (result i32)
    (local i32 i32 i32 i32 i32) ;local variables;
    ...
    loop  ; label = @1 ;
      ...
      i32.eqz
      if  ; label = @2 Compare the top of the stack ;
        ...
        local.get 0
        return ; Return the last element which is saved in local 0 ;
      end
      ...
      block  ;label = @2 ;
        ...
        i32.store ; Store the fib value in the mem assigned to the
            ↪ result array;
        br 1 (;@1;) ;Continue the loop;
      end
    end)
  ...
  (func (;44;) (type 8) (result i32)
    ...
    call 2 ; Calling the imported function to get input ;
    i32.store ; Store the input in memory ;
  ...
  (func (;45;) (type 7)
    (local i32 i32 i32)
    ...
    call 44
    call 40 ; Calling fibo function ;
    i32.store offset=20
  ...
  (table (;0;) 33 33 funcref)
  ; Memory section: 0x05 0x00 0x00 0x00 0x03 ... ;
  (memory (;0;) 17)
  ; Global section: 0x06 0x00 0x00 0x00 0x11.. ;
  (global (;global index 0;) (mut i32 ;mut global;) (i32.const 1048576))
  ...
  ; Export section: 0x07 0x00 0x00 0x00 0x72 ... ;
  (export "memory" (memory 0))
  (export "fibo" (func 40))
  (export "main" (func 45))
  ...
  ; Data section: 0x0d 0x00 0x00 0x03 0xEF  ... ;
  (data (;data segment index 0;) (i32.const 1048576) "invalid args...")
  ...
; Custom section: 0x00 0x00 0x00 0x00 0x2F ;
  (@custom "producers" "...")
```

***Listing 2.2:*** *Refer to Listing 2.1 for the Rust code example. This example showcases the transition from Rust to Wasm, where numerous high-level language attributes convert into multiple Wasm instructions. For clarity, we've marked elements and portions of the WebAssembly binary as comments.*

### 2.1.2 Extending WebAssembly

The broad spectrum of applicability and the rapid adoption of WebAssembly has resulted in demands for additional features. However, not all of these demands align with its original specifications. Thus, since the introduction of WebAssembly, various extensions have been proposed for standardization. For instance, the SIMD proposal enables the execution of vectorized instructions in WebAssembly. To become a standard, a proposal must fulfill certain criteria, including having a formal specification and at least two independent implementations, e.g., two different engines. Notably, even after adoption, new extensions are optional, e.g., the core WebAssembly remains untouched and continues to be referred to as version 1.0.

### 2.1.3 WebAssembly's binary format

The Wasm binary format is close to machine code and already optimized, being a consecutive collection of sections. In Figure 2.1 we show the binary format of a Wasm section. A Wasm section starts with a 1-byte section ID, followed by a 4-byte section size, and concludes with the section content, which precisely matches the size indicated earlier. A WebAssembly binary contains sections of 13 types, each with a specific semantic role and placement within the module. For instance, the *Custom Section* stores metadata like the compiler used to generate the binary, while the *Type Section* contains function signatures that serve to validate the *Function Section*. The *Import Section* lists elements imported from the host, and the *Function Section* details the functions defined within the binary. Other sections like *Table, Memory, and Global Sections* specify the structure for indirect calls, unmanaged linear memories, and global variables, respectively. *Export, Start, Element, Code, Data, and Data Count Sections* handle aspects ranging from declaring elements for host engine access to initializing program state, declaring bytecode instructions per function, and initializing linear memory. Each of these sections must occur only once in a binary and can be empty. For clarity, we also annotate sections as comments in the Wasm code in Listing 2.2.

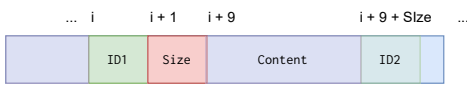



***Figure 2.1:*** *Memory byte representation of a WebAssembly binary section, starting with a 1-byte section ID, followed by an 8-byte section size, and finally the section content.*

A WebAssembly binary can be processed efficiently due to its organization into a contiguous array of sections. For instance, this structure permits compilers to expedite the compilation process either through parallel parsing or by disregarding *Custom Sections*. Moreover, the *Code Section*'s instructions

are further compacted through the use of the LEB128[7] encoding. Consequently, Wasm binaries are not only fast to validate and compile, but also swift to transmit over a network.

### 2.1.4 WebAssembly's runtime

The WebAssembly's runtime characterizes the behavior of WebAssembly programs during execution. This section describes the main components of the WebAssembly runtime, namely the execution stack, functions, memory model, and execution process. These components are crucial for understanding both the WebAssembly's control-flow and the analysis of WebAssembly binaries.

**Execution Stack:** At runtime, WebAssembly engines instantiate a WebAssembly module. This module is a runtime representation of a loaded and initialized WebAssembly binary described in Section 2.1.3. The primary component of a module instance is its Execution Stack. The Execution Stack stores typed values, labels, and control frames. Labels manage block instruction starts and loop starts. Control frames manage function calls and function returns. Values within the stack can only be static types. These types include `i32` for 32-bit signed integers, `i64` for 64-bit signed integers, `f32` for 32-bit floats, and `f64` for 64-bit floats. Abstract types such as classes, objects, and arrays are not supported natively. Instead, these types are abstracted into primitive types during compilation and stored in linear memory.

**Functions:** At runtime, WebAssembly functions are closures over the module instance, grouping locals and function bodies. Locals are typed variables that are local to a specific function invocation. A function body is a sequence of instructions that are executed when the function is called. Each instruction either reads from the execution stack, writes to the execution stack, or modifies the control-flow of the function. Recalling the example WebAssembly binary previously showed, the local variable declarations and typed instructions that are evaluated using the stack can be appreciated between Line 12 and Line 38 in Listing 2.2. Each instruction reads its operands from the stack and pushes back the result. Notice that, numeric instructions are annotated with its corresponding type. In the case of Listing 2.2, the result value of the main function is the calculation of the last instruction, `i32.add` in line 38. As the listing also shows, instructions are annotated with a numeric type.

**Memory model:** A WebAssembly module instance incorporates three key types of memory-related components: linear memory, local variables and global variables. These components can either be managed solely by the host engine or shared with the WebAssembly binary itself. This division of responsibility is often categorized as *managed* and *unmanaged* memory [**?** ]. Managed refers

[7] `https://en.wikipedia.org/wiki/LEB128`

to components that are exclusively modified by the host engine at the lowest level, e.g. when the WebAssembly binary is JITed, while unmanaged components can also be altered through WebAssembly opcodes. First, modules may include multiple linear memory instances, which are contiguous arrays of bytes. These are accessed using 32-bit integers (`i32`) and are shareable only between the initiating engine and the WebAssembly binary. Generally, these linear memories are considered to be unmanaged, e.g., line 21 of Listing 2.2 shows an explicit memory access opcode. Second, there are global instances, which are variables accompanied by values and mutability flags (see example in line 42 of Listing 2.2). These globals are managed by the host engine, which controls their allocation and memory placement completely oblivious to the WebAssembly binary scope. They can only be accessed via their declaration index, prohibiting dynamic addressing. Third, local variables are mutable and specific to a given function instance. They are accessible only through their index relative to the executing function and are part of the data managed by the host engine.

**WebAssembly module execution:** While a WebAssembly binary could be interpreted, the most practical approach is to JIT compile it into machine code. The main reason is that WebAssembly is optimized and closely aligned with machine code, leading to swift JIT compilation for execution. Browser engines such as V8[8] and SpiderMonkey[9] utilize this strategy when executing WebAssembly binaries in browser clients. Once JITed, the WebAssembly binary operates within a sandboxed environment, accessing the host environment exclusively through imported functions. The communication between the host and the WebAssembly module execution is typically facilitated by trampolines in the JITed machine code.

**WebAssembly standalone engines:** While initially intended for browsers, WebAssembly has undergone significant evolution, primarily due to WASI[**?** ]. WASI establishes a standardized POSIX-like interface for interactions between WebAssembly modules and host environments. Compilers can generate WebAssembly binaries that implement WASI, which allows execution in standalone engines. These binaries can then be executed by standalone engines across a variety of environments, including the cloud, servers, and IoT devices [**?** ]. Similarly to browsers, these engines often translate WebAssembly into machine code via JIT compilation, ensuring a sandboxed execution process. Standalone engines such as WASM3[10], Wasmer[11], Wasmtime[12], WAVM[13], and Sledge[**?** ] have been developed to support both WebAssembly and WASI. In a related

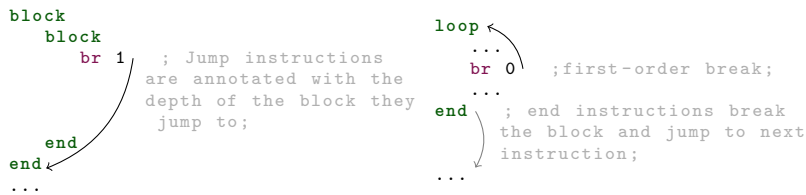
[8] `https://chromium.googlesource.com/v8/v8.git`
[9] `https://spidermonkey.dev/`
[10] `https://github.com/wasm3/wasm3`
[11] `https://wasmer.io/`
[12] `https://github.com/bytecodealliance/wasmtime`
[13] `https://github.com/WAVM/WAVM`

development, Singh et al. [**?** ] have created a WebAssembly virtual machine specifically designed for Arduino-based devices.

### 2.1.5 WebAssembly's control-flow

A WebAssembly function groups instructions into blocks, with the function's entrypoint acting as the root block. In contrast to conventional assembly code, control-flow structures in Wasm leap between block boundaries rather than arbitrary positions within the code, effectively prohibiting `goto`s to random code positions. Each block may specify the needed execution stack state before execution as well as the resultant execution stack state once its instructions have been executed. Typically, the execution stack state is simply the quantity and numeric type of values on the stack. This stack state is used to validate the binary during compilation and to ensure that the stack is in a valid state before the execution of the block's instructions. Blocks in Wasm are explicit (see instructions `block` and `end` in lines 16 and 34 of Listing 2.2), delineating where they commence and conclude. By design, a block cannot reference or execute code from external blocks.

During runtime, WebAssembly break instructions can only jump to one of its enclosing blocks. Breaks, except for those within loop constructions, jump to the block's end and continue to the next immediate instruction. For instance, after line 34 of Listing 2.2, the execution would proceed to line 35. Within a loop, the end of a block results in a jump to the block's beginning, thus restarting the loop. For example, if line 30 of Listing 2.2 evaluates as false, the next instruction to be executed in the loop would be line 18. Listing 2.3 provides an example for better understanding, comparing a standard block and a loop block in a Wasm function.




***Listing 2.3:*** *Example of breaking a block and a loop in WebAssembly.*

Each break instruction includes the depth of the enclosing block as an operand. This depth is used to identify the target block for the break instruction. For example, in the left-most part of the previously discussed listing, a break instruction with a depth of 1 would jump past two enclosing blocks. This design hardens the rewriting of WebAssembly binaries. For instance, if an outer block is removed, the depth of the break instructions within nested blocks must be updated to reflect the new enclosing block depth. This is a significant challenge for

rewriting tools, as it requires the analysis of the control-flow graph to determine the enclosing block depth for each break instruction.

### 2.1.6 Security and Reliability for WebAssembly

The WebAssembly ecosystem's expansion needs robust tools to ensure its security and reliability. Numerous tools, employing various strategies to detect vulnerabilities in WebAssembly programs, have been created to meet this need. This paper presents a review of the most relevant tools in this field, focusing on those capable of providing security guarantees for WebAssembly binaries.

**Static analysis:** SecWasm[**?** ] uses information control-flow strategies to identify vulnerabilities in WebAssembly binaries. Conversely, Wasmati[**?** ] employs code property graphs for this purpose. Wasp[**?** ] leverages concolic execution to identify potential vulnerabilities in WebAssembly binaries. VeriWasm[**?** ], an offline verifier designed specifically for native x86-64 binaries JITed from WebAssembly, adopts a unique approach. While these tools emphasize specific strategies, others adopt a more holistic approach. CT-Wasm[**?** ], verifies the implementation of cryptographic algorithms in WebAssembly. Similarly, Vivienne applies relational Symbolic Execution (SE) to WebAssembly binaries in order to reveal vulnerabilities in cryptographic implementations[**?** ]. For example, both Wassail[**?** ] and WasmA[**?** ] provide a comprehensive static analysis framework for WebAssembly binaries. However, static analysis tools may have limitations. For instance, a newly, semantically equivalent WebAssembly binary may be generated from the same source code bypassing or breaking the static analysis [**?** ]. If the WebAssembly input differs from the input used during sound analysis [**?** ], the vulnerability may go unnoticed. Thus, there may be a lack of subjects to evaluate the effectiveness of these tools.

**Dynamic analysis:** Dynamic analysis involves tools such as TaintAssembly[**?** ], which conducts taint analysis on WebAssembly binaries. Fuzzm[**?** ] identifies vulnerabilities in host engines by conducting property fuzzing through WebAssembly binary execution. Furthermore, Stiévenart and colleagues have developed a dynamic approach to slicing WebAssembly programs based on Observational-Based Slicing (ORBS)[**?** **?** ]. This technique aids in debugging, understanding programs, and conducting security analysis. However, Wasabi[**?** ] remains the only general-purpose dynamic analysis tool for WebAssembly binaries, primarily used for profiling, instrumenting, and debugging WebAssembly code. Similar to static analysis, these tools typically analyze software behavior during execution, making them inherently reactive. In other words, they can only identify vulnerabilities or performance issues while pseudo-executing input WebAssembly programs. Thus, facing an important limitation on overhead for real-world scenarios.

**Protecting WebAssembly binaries and runtimes:** The techniques discussed previously are primarily focused on reactive analysis of WebAssembly binaries. However, there exist approaches to harden WebAssembly binaries, enhancing their secure execution, and fortifying the security of the entire execution runtimes ecosystem. For instance, Swivel[**?** ] proposes a compiler-based strategy designed to eliminate speculative attacks on WebAssembly binaries, particularly in Function-as-a-Service (FaaS) platforms such as Fastly. Similarly, Kolosick and colleagues [**?** ] modify the Lucet compiler to use zero-cost transitions, eliminating thee performance overhead of SFI guarantees implementation. Conversely, WaVe[**?** ] introduces a mechanized engine for WebAssembly that facilitates differential testing. WaVe can be employed to detect anomalies in engines running Wasm-WASI programs. Much like static and dynamic analysis tools, these tools may suffer from a lack of WebAssembly inputs, which could affect the measurement of their effectiveness.

**WebAssembly malware:** Since the introduction of WebAssembly, the Web has consistently experienced an increase in cryptomalware. This rise primarily stems from the shift of mining algorithms from CPUs to WebAssembly, a transition driven by notable performance benefits [**?** ]. Tools such as MineSweeper[**?** ], MinerRay[**?** ], and MINOS[**?** ] employ static analysis with machine learning techniques to detect browser-based cryptomalwares. Conversely, SEISMIC[**?** ], RAPID[**?** ], and OUTGuard[**?** ] leverage dynamic analysis techniques to achieve a similar objective. VirusTotal[14], a tool incorporating over 60 commercial antivirus systems as black-boxes, is capable of detecting cryptomalware in WebAssembly binaries. However, obfuscation studies have exposed their shortcomings, revealing an almost unexplored area for WebAssembly that threatens malware detection accuracy. In concrete, Bahnsali et al. seminal work[**?** ] demonstrate that cryptomining algorithm's source code can evade previous techniques through the use of obfuscation techniques.

### 2.1.7 Open challenges

Despite progress in WebAssembly analysis, numerous challenges remain. WebAssembly, though deterministic and well-typed by design, is susceptible to a variety of security threats. First, most existing WebAssembly research is reactive, focusing on detecting and fixing vulnerabilities already reported. This approach leaves WebAssembly binaries and runtime implementations potentially open to unidentified attacks. Second, side-channel attacks present a significant risk. Genkin et al., for example, illustrated how WebAssembly could be manipulated to extract data via cache timing-side channels [**?** ]. Furthermore, research conducted by Maisuradze and Rossow demonstrated the potential for speculative execution attacks on WebAssembly binaries [**?** ]. Rokicki et al. disclosed the possibility

[14] `https://www.virustotal.com`

for port contention side-channel attacks on WebAssembly binaries in browsers [**?** ]. Finally, the binaries themselves may be inherently vulnerable. For example, studies by Lehmann et al. and Stiévenart et al. suggested that flaws in C/C++ source code could infiltrate WebAssembly binaries [**?** **?** ].

## 2.2 Software diversification

Software diversification involves the synthesis, reuse, distribution, and execution of different, functionally equivalent programs. As outlined in Baudry et al.'s survey [**?** ], software diversification falls into five usage categories: reusability [**?** ],performance [**?** ], fault tolerance [**?** ], software testing [**?** ], and security [**?** ]. Our work specifically contributes to the last two categories. Based on the works of Cohen et al. [**?** ], Forrest et al. [**?** ], Jackson et al. [**?** ] and Baudry et al. [**?** ], this section presents core concepts and related works, emphasizing how they generate diversification and apply it to WebAssembly.

### 2.2.1 Generation of Software Variants

Software variants are functionally equivalent versions of an original program, created through software diversification at different stages of the software lifecycle, such as the source code or machine code levels. The diversification can be either natural [**?** ] or artificial [**?** ].

**Natural Diversity:** Natural diversity denotes the innate process wherein humans create software variants using various programming languages, compilers, and operating systems [**?** ], all adhering to the same initial functional requirements. For instance, Firefox and Chrome web browsers exemplify natural diversity, given their practical differences, they serve the same purpose. Natural diversity plays a crucial role in securing systems, as different variants are not susceptible to identical vulnerabilities. Creating natural software variants demands a significant amount of human effort and time. This makes it impractical for new ecosystems such as WebAssembly. On the other hand, humans are not always the best at creating diverse software variants since they think alike [**?** ]. Thus, systematic and artificial diversification approaches are better.

**Artificial Diversity:** The concept of artificial software variants starts with Randell's 1975 work [**?** ], which put forth the notion of artificial fault-tolerant instruction blocks. Artificial software diversification, as proposed by Cohen and Forrest in the 1990s [**?** **?** ], gets its development through rewriting strategies. These strategies consist of rule sets for modifying software components to create functionally equivalent, yet distinct, programs. Rewriting strategies typically take the form of tuples: `instr1 => (instr2, instr3, ...)`, where `instr` represents the original code and `(instr2, instr3, ...)` denotes the functionally equivalent code. This dissertation focuses on artificial software

diversification, as it is more practical, systematic, and scalable than natural diversity.

**Rewriting strategy:** The creation of artificial software diversification commences with rewriting rules. A rewriting rule refers to a functionally equivalent substitution for a code segment, manually written. These rules can be applied at varying levels, from coarse to fine-grained. This can range from the program dependencies level [**?** ] to the instruction level [**?** ]. For example, Cleemput et al. [**?** ] and Homescu et al. [**?** ] inject NOP instructions to yield statically varied versions at the instruction level. Here, the rewriting rule is represented as `instr => (nop instr)`, signifying a `nop` operation preceding the instruction.

**Instruction Reordering:** This strategy reorders instructions in a program. For example, variable declarations may change if compilers reorder them in the symbol tables. This prevents static examination and analysis of parameters and alters memory locations. In this area, Bhatkar et al. [**? ?** ] proposed the random permutation of variable and routine order for ELF binaries. Such strategies are not implemented for WebAssembly to the best of our knowledge.

**Adding, Changing, Removing Jumps and Calls:** This strategy generates program variants by adding, changing, or removing jumps and calls in the original program. Cohen [**?** ] primarily illustrated this concept by inserting random jumps in programs. Pettis and Hansen [**?** ] suggested splitting basic blocks and functions for the PA-RISC architecture, inserting jumps between splits. Similarly, Crane et al. [**?** ] de-inlined basic blocks of code as an LLVM pass. In their approach, each de-inlined code transforms into semantically equivalent functions that are randomly selected at runtime to replace the original code calculation. On the same topic, Bhatkar et al. [**?** ] extended their previous approach [**?** ], replacing function calls with indirect pointer calls in C source code, allowing post-binary reordering of function calls. In the WebAssembly context, the most analogous work is wobfuscator [**?** ]. Wobfuscator, a JavaScript obfuscator, substitutes JavaScript code with WebAssembly code, e.g., numeric calcula. This strategy effectively uses the interleaving of calls between JavaScript and WebAssembly to provide JavaScript variants.

**Program Memory and Stack Randomization:** This strategy alters the layout of programs in the host memory. Additionally, it can randomize how a program variant operates its memory. The work of Bhatkar et al. [**? ?** ] proposes to randomize the base addresses of applications and library memory regions in ELF binaries. Tadesse Aga and Autin [**?** ], and Lee et al. [**?** ] propose a technique to randomize the local stack organization for function calls using a custom LLVM compiler. Younan et al. [**?** ] suggest separating a conventional stack into multiple stacks where each stack contains a particular class of data. On the same topic, Xu et al. [**?** ] transforms programs to reduce memory exposure time, improving the time needed for frequent memory address randomization. This makes it very

challenging for an attacker to ignore the key to inject executable code. This strategy disrupts the predictability of program execution and mitigates certain exploits such as speculative execution. No work has been found that explicitly applies this strategy to WebAssembly.

**ISA Randomization and Simulation:** This strategy involves using a key to cypher the original program binary into another encoded binary. Once encoded, the program can only be decoded at the target client, or it can be interpreted in the encoded form using a custom virtual machine implementation. This technique is strong against attacks involving code inspection. Kc et al. [**?** ], and Barrantes et al. [**?** ] proposed seminal works on instruction-set randomization to create a unique mapping between artificial CPU instructions and real ones. On the same topic, Chew and Song [**?** ] target operating system randomization. They randomize the interface between the operating system and the user applications. Couroussé et al. [**?** ] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks. Their technique generates a different program during execution using an interpreter for their DSL. Generally, *ISA randomization and simulation* usually faces a performance penalty, especially for WebAssembly, due to the decoding process as shown in WASMixer evaluation [**?** ].

**Code obfuscation:** Code obfuscation can be seen as a simplification of *ISA randomization*. The main difference between encoding and obfuscating code is that the former requires the final target to know the encoding key while the latter executes as is in any client. Yet, both strategies aim to tackle static reverse engineering of programs. In the context of WebAssembly, Romano et al. [**?** ] proposed an obfuscation technique, wobfuscator, for JavaScript in which part of the code is replaced by calls to complementary WebAssembly functions. Yet, wobfuscator targets JavaScript code, not WebAssembly binaries.

**Enumerative synthesis:** Enumerative synthesis is a fully automated and systematic approach to generate program variants. It examines all possible programs specific to a given language. The process of enumerative synthesis commences with a piece of input program, typically a basic block. Incrementally, using a defined grammar, it generates all programs of size $n$. A generated program is then checked for equivalence to the original program, either by using a test suite or a theorem solver. If the generated variant is proved, it is added to the variants collection. The procedure continues until all potential programs have been explored. This approach proves especially effective when the solution space is relatively small or can be navigated efficiently. Jacob and colleagues [**?** ] implemented this strategy for x86 programs. They named this technique superdiversification, drawing parallels to superoptimization [**?** ]. Since this strategy fully explores a program's solution space, it contains the aforementioned strategies as special cases. The application of enumerative synthesis to WebAssembly has not been explored.

**TODO** TBD: add one on AI driven ?

### 2.2.2 Equivalence Checking

Equivalence checking between program variants is a vital component for any program transformation task, ranging from checking compiler optimizations [**?** ] to the artificial synthesis of programs discussed in this chapter. It proves that two pieces of code or programs are functionally equivalent [**?** ]. We can roughly simplify the checking process with the following property: two programs are deemed equivalent if they generate identical outputs. This equivalence is observed when given identical inputs from a closed collection of inputs [**?** ]. We adopt this definition of *functional equivalence modulo input* throughout this dissertation. In Software Diversification, equivalence checking seeks to preserve the original functionality of programs while varying observable behaviors. Two programs, for instance, can differ statically and still compute the same result. We outline two methods to check variant equivalence: by construction and prove-driven equivalence checking.

**Equivalence checking by construction:** The equivalence property is often guaranteed by construction. Cleemput et al. [**?** ] and Homescu et al. [**?** ], for example, design their transformation strategies to generate semantically equivalent program variants. However, developer errors can occur in this process, necessitating further validation. The test suite of the original program can serve as a check for the variant. If the program variant passes the test suite [**?** ], it can be considered equivalent to the original. However, this technique is limited by the need for a preexisting test suite and does not give guarantees. An alternative method for checking program equivalence involves the use of fuzzers [**?** ]. Fuzzers randomly generate inputs that yield different observable behaviors. If two inputs produce a different output in the variant, the variant and the original program are not equivalent. The primary limitation for fuzzers is that the process is notably time-consuming and necessitates manual introduction of oracles. Recent advances in machine learning have prompted researchers to investigate the use of neural networks for verifying program equivalence. One such example is the work of Zhang and colleagues [**?** ], which produces reference oracles and test cases by using Large Language Models. Although this method appears effective, it only achieves an accuracy rate of 88%, which is insufficient for security applications and total verification.

**Prove-driven checking:** In the absence of a test suite or a technique that inherently implements the equivalence property, the works mentioned earlier use theorem solvers (SMT solvers) [**?** ] to prove equivalence of program variants. The central idea for SMT solvers is to convert the two code variants into mathematical formulas. The SMT solver then checks for counter-examples. When it finds a counter-example, there is an input for which the two mathematical

formulas yield different outputs. The primary limitation of this technique is that not all algorithms can be translated into a mathematical formula, such as loops. Nevertheless, this technique is frequently used for checking no-branching-programs like basic block and peephole replacements [**?** ].

### 2.2.3 Variants deployment

Program variants, once generated and verified, may be utilized in two primary scenarios: Randomization or Multivariant Execution (MVE) [**?** ].

**Randomization:** In the context of our work, the term *Randomization* denotes a program's ability to present different variants to different clients. In this setup, a program, chosen from a collection of variants (referred to as the program's variant pool), is assigned to a random client during each deployment. Jackson et al. [**?** ] define the variant pool in Randomization as herd immunity, as vulnerable binaries can only affect a segment of the client community. El-Khalil and colleagues [**?** ] suggest employing a custom compiler to generate varying binaries from the compilation process. They adapt a version of GCC 4.1 to partition a conventional stack into several component parts, termed multistacks. Similarly, Singhal and colleagues, propose Cornucopia [**?** ]. Cornucopia generates multiple variants of a program by using different compiler flag combinations. Aga and colleagues [**?** ], contributing to this discussion, propose the generation of program variants through the randomization of its data layout in memory. This method allows each variant to operate on the same data in memory but at different memory offsets. Randomization can also be applied to virtual machines and operating systems. On this note, Kc et al. [**?** ] establish a unique mapping between artificial CPU instructions and actual ones, enabling the assignment of various variants to specific target clients. In a similar vein, Xu et al. [**?** ] recompile the Linux Kernel to minimize the exposure time of persistent memory objects, thereby increasing the frequency of address randomization.

**Multivariant Execution (MVE):** Multiple program variants are composed into a single binary, known as a multivariant binary [**?** ]. Each multivariant binary is randomly deployed to a client. Then, the multivariant binary executes its embedded program variants at runtime. These embedded variants can either execute in parallel to check for inconsistencies, or as a single program to randomize execution paths [**?** ]. Bruschi and colleagues extend the concept of executing two variants in parallel, introducing non-overlapping and randomized memory layouts [**?** ]. At the same time, Salamat et al. modifies a standard library to generate 32-bit Intel variants. These variants have a stack that grows in the opposite direction, allowing for the detection of memory inconsistencies [**?** ]. Davi and colleagues propose Isomeron, an approach for execution-path randomization [**?** ]. Isomeron operates by simultaneously loading the original program and a variant. It then uses a coin flip to determine which copy of

the program to execute next at the function call level. Previous works have highlighted the benefits of limiting execution to only two variants in a multivariant environment. Agosta and colleagues, as well as Crane and colleagues, used more than two generated programs in the multivariant composition, thereby randomizing software control flow at runtime [**?** **?** ]. Both strategies have proven effective in enhancing security by addressing known vulnerabilities, such as Just-In-Time Return-Oriented Programming (JIT-ROP) attacks [**?** ] and power side-channel attacks [**?** ]. Lastly, only Voulimeneas et al. [**?** ] have recently proposed a multivariant execution system that enhances security by parallelizing the execution of variants across different machines.

### 2.2.4 Software Diversification Asssessment

Assessing software diversification presents a significant challenge. The size of the variant space does not necessarily correlate with a variant's capacity to fulfill an objective such as hardening attacks [**?** ]. Ideally, real scenarios would provide the most accurate assessment of diversification, e.g., demonstrating a variant's effectiveness under specific attacks. However, such an approach is not always feasible, e.g., as previously discussed, Software Diversification is a preemptive strategy. Hence, a combination of metrics is required for the creation and assessment of software diversification.

**Static comparison of variants:** Static metrics are used to evaluate and identify the diversity of programs without needing execution. The fundamental concept entails comparing variant source codes or binary codes to determine how diverse they are. Usually, comparing variants means defining a distance metric between programs. At the low-level of bytecode instructions, for example, these metrics include Levenshtein distance [**?** ], and global alignments [**?** ]. On the other hand, at the high-level of source code, these metrics often rely on Abstract Syntax Tree (AST) diffing, such as GUMtree-based distances [**?** ]. Bostani et al. [**?** ] illustrate the use of static distances in guiding the generation process of variants. They categorize the space of Android applications into malware and goodware. Then, they create malware variants by employing a static distance metric to approach the goodware group as closely as possible, thus successfully evading malware classifiers.

**Dynamic comparison of variants:** Static comparisons between variants inherently have limitations. For example, two variants may show differences at the source code level but exhibit identical behavior during execution. Take the addition of `nop` operations to a program as an instance. Despite source code level differences, the variant and the original program execute identical instructions, leading to similar behaviors modulo input. Assessing Software Diversification primarily aims to demonstrate variant-specific observabilities. While static differences are observable, runtime information holds complementary

relevance [**?** ]. Therefore, dynamic metrics are essential to assess the diversity of variants. For instance, Forrest et al. [**?** ] were pioneers in classifying program behaviors by analyzing their system call traces using n-grams profiling. Cabrera et al. utilized a global alignments approach to gauge the diversity of JavaScript bytecode traces within the Chrome browser [**?** ]. Fang et al. proposed a method to counteract JavaScript obfuscation techniques used in malicious code, by analyzing dynamic information captured from V8 bytecode traces [**?** ]. Dynamic metrics are primarily employed to cluster similar behaviors. Yet, they assess the diversity of program variants. i.e., the diversity during runtime is greater when the difference between two behaviors is larger. Notice that, dynamic assessment can be difficult due to the expense of program execution or the complication of required user interaction. On the other hand, malware programs, which usually do not require user interaction, are simpler to evaluate in controlled environments before actual deployment.

In the context of WebAssembly, there exist no explicit works on Software Diversification. Consequently, previous metrics have not been directly applied to assess diversification in WebAssembly binaries. However, in other domains, such as the analysis of WebAssembly binaries, several studies have employed static metrics. For example, VeriWasm quantifies attack-based patterns, stating that a WebAssembly binary is more secure with a lower pattern count [**?** ]. This metric might potentially serve as a guide during variant generation. In the field of malware detection, MINOS [**?** ] proposes transforming WebAssembly binaries into grayscale images. They then employ convolutional neural networks to identify malware, where an increased similarity to a malware image increases the probability of the binary being malware. Regarding the dynamic comparisons, Wang et al.'s study [**?** ] profiles WebAssembly instructions during runtime to identify malicious behavior.

### 2.2.5 Offensive Diversification

Lundquist and colleagues [**?** ] distinguish Software Diversification into two categories: Defensive and Offensive Diversification. On the one hand, Defensive Software Diversification introduces unpredictability in system behavior. By making software less predictable, defensive software diversification aims to proactively deter attacks, acting as a complementary strategy to other, more reactive, security measures. The majority of previously discussed works in this section contribute to defensive diversification. Yet, Software Diversification that aims to create diverse harmful programs is considered Offensive Diversification [**?** ].

**Offensive Diversification:** Offensive Diversification is conceptually equal to Defensive Software Diversification. Yet, in an offensive context, one may apply diversification techniques to malware or other malicious codes to evade detection

by security software [**?** ]. For example, one might equate Offensive Diversification with Code obfuscation, if its purpose shifts from preventing reverse engineering by malicious actors, to evading detection by malware analysis systems.

Malicious actors may employ previously discussed diversification strategies to evade detection, including genetic programming [**?** ]. Over time, these evasion techniques have evolved in both complexity and sophistication [**?** ]. Chua et al. [**?** ], for instance, suggested a framework for automatically obfuscating the source code of Android applications using method overloading, opaque predicates, try-catch, and switch statement obfuscation, resulting in multiple versions of identical malware. Moreover, machine learning approaches have been utilized to develop evasive malware [**?** ], drawing on a corpus of pre-existing malware [**?** ]. These methods aim to thwart static malware detectors, yet, more advanced techniques focus on evading dynamic detection mostly by employing throttling [**? ?** ].

The term Offensive Software Diversification may appear counterintuitive. Yet, such approaches measure the resilience and accuracy of security systems. This is an almost unexplored area in WebAssembly, posing a threat to malware detection accuracy. Specifically, only Bahnsali et al.'s seminal work[**?** ] has demonstrated that a cryptomining algorithm's source code can evade pre-existing malware detection methods. More recently, Madvex [**?** ] has sought to obfuscate WebAssembly binaries to achieve malware evasion, but this approach is limited to altering only the code section of WebAssembly binaries.

### 2.2.6 Open challenges

As outlined in Section 2.1.7, our primary motivation for the contributions of this thesis is the open issues within the WebAssembly ecosystem. We see potential in employing Software Diversification to address them. Based on our previous discussion, we highlight several open challenges in the realm of Software Diversification for WebAssembly. First, WebAssembly, being an emerging technology, is still in the process of implementing defensive measures [**?** ]. The process of officially adopting a new defensive measure is inherently slow, making software diversification a potentially valuable preemptive strategy. Second, despite the abundance of related work on software diversity, its exploration in the context of WebAssembly remains limited. Third, both randomization and multivariant execution have been largely unexplored. Lastly, the works on malware detection discussed in Section 2.1.6 suggest that offensive diversification could be useful in measuring the resilience and accuracy of security systems for WebAssembly.

## Conclusions

In this chapter, we presented an overview of the Wasm language. This included its binary format, runtime execution concepts, and security issues. Related work was also discussed. The goal of this chapter is to establish a foundation for studying automatic diversification in Wasm. We emphasized the fact that Wasm has not been extensively researched in the field of Artificial Software Diversification. Existing implementations for Software Diversification cannot be directly applied to Wasm. Current security limitations and the absence of software diversity approaches for Wasm inspire our work. In Chapter 3, we elaborate on the technical details that guide our contributions.

# 3
# AUTOMATIC SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

*All problems in computer science can be solved by another level of indirection, except for the problem of too many layers of indirection.*

— David Wheeler

**TODO** Map concepts with chapter2

THE process of generating WebAssembly binaries starts with the original source code, which is then processed by a compiler to produce a WebAssembly binary. This compiler is generally divided into three main components: a frontend that converts the source code into an intermediate representation, an optimizer/transformer that modifies this representation usually for performance, and a backend that compiles the final WebAssembly binary. This architecture is illustrated in the left most part of Figure 3.1.

Software Diversification, a preemptive security measure, can be integrated at various stages of this compilation process. However, applying diversification at the front-end has its limitations, as it would need a unique diversification mechanism for each language compatible with the frontend component. Conversely, diversification at later compiler stages, such as the optimizer or backend, offers a more practical alternative. This makes the latter stages of the compilers an ideal point for introducing practical Wasm diversification techniques. Our compiler-based strategies, represented in red and green in Figure 3.1, introduce a diversifier component into the optimizer/transformer and backend stages. This optimization/transformer component generates variants in the intermediate representation of a compiler, thereby creating artificial software diversity for WebAssembly. The variants are then compiled into WebAssembly binaries by the backend component of the compiler. Specifically, we propose two tools: CROW, which generates WebAssembly program variants, and MEWE,

[0] Compilation probe time 2023/11/07 12:55:02

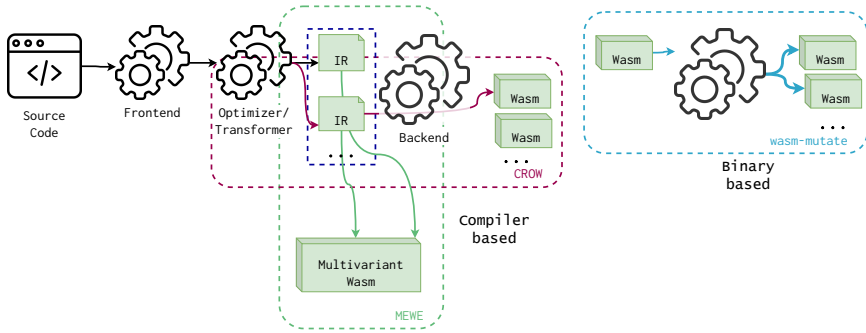



***Figure 3.1:*** *Approach landscape containing our three technical contributions: CROW squared in red, MEWE squared in green and WASM-MUTATE squared in blue. We annotate where our contributions, compiler-based and binary-based, stand in the landscape of generating WebAssembly programs.*

which packages these variants to enable multivariant execution [**?** ]. Alternatively, diversification can be directly applied to the WebAssembly binary, offering a language and compiler-agnostic approach. Our binary-based strategy, WASM-MUTATE, represented in blue in Figure 3.1, employs rewriting rules on an e-graph data structure to generate a variety of WebAssembly program variants.

This dissertation contributes to the field of Software Diversification for WebAssembly by presenting two primary strategies: compiler-based and binary-based. Within this chapter, we introduce three technical contributions: CROW, MEWE, and WASM-MUTATE. We also compare these contributions, highlighting their complementary nature. Additionally, we provide the artifacts for our contributions to promote open research and reproducibility of our main takeaways.

## 3.1 CROW: Code Randomization of WebAssembly

This section details CROW [**?** ], represented as the red squared tooling in Figure 3.1. CROW is designed to produce functionally equivalent Wasm variants from the output of an LLVM front-end, utilizing a custom Wasm LLVM backend.

Figure 3.2 illustrates CROW's workflow in generating program variants, a process compound of two core stages: *exploration* and *combination*. During the *exploration* stage, CROW processes every instruction within each function of the LLVM input, creating a set of functionally equivalent code variants. This process ensures a rich pool of options for the subsequent stage. In the *combination* stage, these alternatives are assembled to form diverse LLVM IR variants, a task achieved through the exhaustive traversal of the power set of all potential combinations of code replacements. The final step involves the custom

Wasm LLVM backend, which compiles the crafted LLVM IR variants into Wasm binaries.

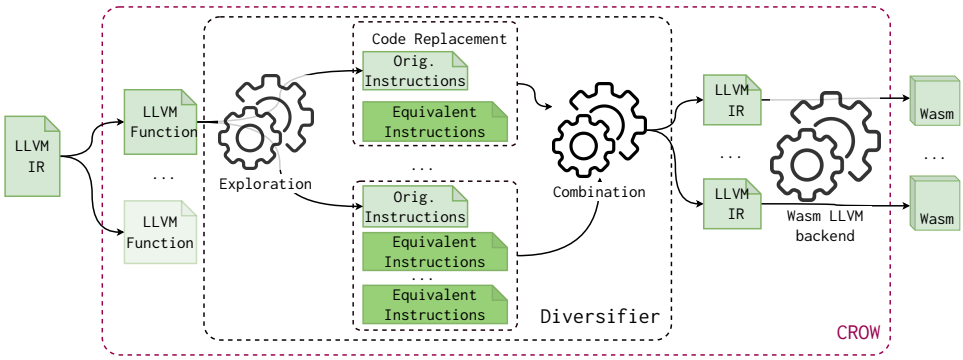



***Figure 3.2:*** *CROW components following the diagram in Figure 3.1. CROW takes LLVM IR to generate functionally equivalent code replacements. Then, CROW assembles program variants by combining them. Figure taken from [**?** ].*

### 3.1.1 Enumerative synthesis

The cornerstone of CROW's exploration mechanism is its code replacement generation strategy, which is inspired by the superdiversifier methodology proposed by Jacob et al. [**?** ]. The search space for generating variants is delineated through an enumerative synthesis process (see Enumerative synthesis in Section 2.2.1 ), which systematically produces all possible code replacements for each instruction in the original program. If a code replacement is identified to perform identically to the original program, it is reported as a functionally equivalent variant. This equivalence is confirmed using a theorem solver for rigorous verification.

Concretely, CROW is developed by extending the enumerative synthesis implementation found in Souper [**?** ], an LLVM-based superoptimizer. Specifically, CROW constructs a Data Flow Graph for each LLVM instruction that returns an integer. Subsequently, it generates all viable expressions derived from a selected subset of the LLVM Intermediate Representation language for each DFG. The enumerative synthesis process incrementally generates code replacements, starting with the simplest expressions (those composed of a single instruction) and gradually increasing in complexity. The exploration process continues either until a timeout occurs or the size of the generated replacements exceeds a predefined threshold.

Notice that the search space increases exponentially with the size of the language used for enumerative synthesis. To mitigate this issue, we prevent CROW from synthesizing instructions without correspondence in the Wasm

backend, effectively reducing the searching space. For example, creating an expression having the `freeze` LLVM instructions will increase the searching space for instruction without a Wasm's opcode in the end.

CROW is carefully designed to boost the generation of variants as much as possible. First, we disable the majority of the pruning strategies. Instead of preventing the generation of commutative operations during the searching, CROW still uses such transformation as a strategy to generate program variants. Second, CROW applies code transformations independently. For instance, if a suitable replacement is identified that can be applied at $N$ different locations in the original program, CROW will generate $2^N$ distinct program variants, i.e., the power set of applying the transformation or not to each location. This approach leads to a combinatorial explosion in the number of available program variants, especially as the number of possible replacements increases.

Leveraging the ascending nature of its enumerative synthesis process, CROW is capable of creating variants that may outperform the original program in both size and efficiency. For instance, the first functionally equivalent transformation identified is typically the most optimal in terms of code size. This approach offers developers a range of performance options, allowing them to balance between diversification and performance without compromising the latter.

The last stage at CROW involves a custom Wasm LLVM backend, which generates the Wasm programs. For it, we remove all built-in optimizations in the LLVM backend that could reverse Wasm variants, i.e., we disable all optimizations in the Wasm backend that could reverse the CROW transformations.

### 3.1.2 Constant inferring

CROW inherently introduces a novel transformation strategy called *constant inferring*, which significantly expands the variety of WebAssembly program variants. Specifically, CROW identifies segments of code that can be simplified into a single constant assignment, with a particular focus on variables that control branching logic. After applying this *constant inferring* technique, the resulting program diverges substantially from the original program structure. This is crucial for diversification efforts, as one of the primary objectives is to create variants that are as distinct as possible from the original source code [**?** ]. In essence, the more divergent the variant, the more challenging it becomes to trace it back to its original form.

Let us illustrate the case with an example. The Babbage problem code in Listing 3.1 is composed of a loop that stops when it discovers the smallest number that fits with the Babbage condition in Line 4.

```
1   int babbage() {
2       int current = 0,
3           square;
4       while ((square=current*current) %
            ↪ 1000000 != 269696) {
5           current++;
6       }
7       printf ("The number is %d\n",
            ↪ current);
8       return 0 ;
9   }
```

***Listing 3.1:*** *Babbage problem. Taken from [**?** ].*

```
int babbage() {
    int current = 25264;

    printf ("The number is %d\n", current)
        ↪ ;
    return 0 ;
}
```

***Listing 3.2:*** *Constant inferring transformation over the original Babbage problem in Listing 3.1. Taken from [**?** ].*

CROW deals with this case, generating the program in Listing 3.2. It infers the value of `current` in Line 2 such that the Babbage condition is reached [1]. Therefore, the condition in the loop will always be false. Then, the loop is dead code and is removed in the final compilation. The new program in Listing 3.2 is remarkably smaller and faster than the original code. Therefore, it offers differences both statically and at runtime[2]

### 3.1.3 Exemplifying CROW

Let us illustrate how CROW works with the example code in Listing 3.3. The `f` function calculates the value of $2 * x + x$ where `x` is the input for the function. CROW compiles this source code and generates the intermediate LLVM bitcode in the left most part of Listing 3.4. CROW potentially finds two integer returning instructions to look for variants, as the right-most part of Listing 3.4 shows.

```
1  int f(int x) {
2      return 2 * x + x;
3  }
```

***Listing 3.3:*** *C function that calculates the quantity $2x + x$.*

[1] In theory, this value can also be inferred by unrolling the loop the correct number of times with the LLVM toolchain. However, standard LLVM tools cannot unroll the **`while`**-loop because the loop count is too large.

[2] Notice that for the sake of illustration, we show both codes in C language, this process inside CROW is performed directly in LLVM IR.

```
define i32 @f(i32) {
  %2 = mul nsw i32 %0,2
  %3 = add nsw i32 %0,%2

  ret i32 %3
}

define i32 @main() {
  %1 = tail call i32 @f(
       i32 10)
  ret i32 %1
}
```

```
Replacement candidates
      for code_1

%2 = mul nsw i32 %0,2

%2 = add nsw i32 %0,%0

%2 = shl nsw i32 %0, 1:i32
```

```
Replacement candidates for
      code_2

%3 = add nsw i32 %0,%2

%3 = mul nsw %0, 3:i32
```

***Listing 3.4:*** *LLVM's intermediate representation program, its extracted instructions and replacement candidates. Gray highlighted lines represent original code, green for code replacements.*

```
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2

%2 = add nsw i32 %0,%0
%3 = add nsw i32 %0,%2

%2 = shl nsw i32 %0, 1:i32
%3 = add nsw i32 %0,%2
```

```
%2 = mul nsw i32 %0,2
%3 = mul nsw %0, 3:i32

%2 = add nsw i32 %0,%0
%3 = mul nsw %0, 3:i32

%2 = shl nsw i32 %0, 1:i32
%3 = mul nsw %0, 3:i32
```

***Listing 3.5:*** *Candidate code replacements combination. Orange highlighted code illustrate replacement candidate overlapping.*

CROW, detects `code_1` and `code_2` as the enclosing boxes in the left most part of Listing 3.4 shows. CROW synthesizes $2 + 1$ candidate code replacements for each code respectively as the green highlighted lines show in the right most parts of Listing 3.4. The baseline strategy of CROW is to generate variants out of all possible combinations of the candidate code replacements, *i.e.,* uses the power set of all candidate code replacements.

In the example, the power set is the cartesian product of the found candidate code replacements for each code block, including the original ones, as Listing 3.5 shows. The power set size results in 6 potential function variants. Yet, the generation stage would eventually generate 4 variants from the original program. CROW generated 4 statically different Wasm files, as Listing 3.6 illustrates. This gap between the potential and the actual number of variants is a consequence of the redundancy among the bitcode variants when composed into one. In other words, if the replaced code removes other code blocks, all possible combinations having it will be in the end the same program. In the example case, replacing `code_2` by `mul nsw %0, 3`, turns `code_1` into dead code, thus, later replacements generate the same program variants. The rightmost part of Listing 3.5 illustrates how for three different combinations, CROW produces the same variant. We call this phenomenon a *code replacement overlapping.*

```
func $f (param i32) (result i32)
   local.get 0
   i32.const 2
   i32.mul
   local.get 0
   i32.add
```

```
func $f (param i32) (result i32)
   local.get 0
   i32.const 1
   i32.shl
   local.get 0
   i32.add
```

```
func $f (param i32) (result i32)
   local.get 0
   local.get 0
   i32.add
   local.get 0
   i32.add
```

```
func $f (param i32) (result i32)
   local.get 0
   i32.const 3
   i32.mul
```

***Listing 3.6:*** *Wasm program variants generated from program Listing 3.3.*

**Contribution paper and artifact**

CROW is a compiler-based approach. It leverages enumerative synthesis to generate functionally equivalent code replacements and assembles them into diverse Wasm program variants. CROW uses SMT solvers to guarantee functional equivalence.
CROW is fully presented in Cabrera-Arteaga et al. "CROW: Code Randomization of WebAssembly" *at proceedings of Measurements, Attacks, and Defenses for the Web (MADWeb), NDSS 2021* `https://doi.org/10.14722/madweb.2021.23004`.

CROW source code is available at `https://github.com/ASSERT-KTH/slumps`

## 3.2 MEWE: Multi-variant Execution for WebAssembly

This section describes MEWE [**?** ]. MEWE synthesizes diversified function variants by using CROW. It then provides execution-path randomization in a Multivariant Execution (MVE) [**?** ]. Execution path randomization is a technique that randomizes the execution path of a program at runtime, i.e. at each invocation of a function, a different variant is executed [**?** ]. MEWE generates application-level multivariant binaries without changing the operating system or Wasm runtime. It creates an MVE by intermixing functions for which CROW generates variants, as illustrated by the green square in Figure 3.1. MEWE inlines function variants when appropriate, resulting in call stack diversification at runtime.

As illustrated in Figure 3.3, MEWE takes the LLVM IR variants generated by CROW's diversifier. It then merges LLVM IR variants into a Wasm multivariant. In the figure, we highlight the two components of MEWE, *Multivariant Generation* and the *Mixer*. In the *Multivariant Generation* process, MEWE gathers the LLVM IR variants created by CROW. The Mixer component, on the other hand, links the multivariant binary and creates a new entrypoint for the binary called *entrypoint tampering*. The tampering is needed in case the output of CROW are variants of the original entrypoint, e.g. the *main* function. Concretely, it wraps the dispatcher for the entrypoint variants as a new function for the final Wasm binary and is declared as the application entrypoint. The random generator is needed to perform the execution-path randomization. For the random generator, we rely on WASI's specification [**?** ] for the random behavior of the dispatchers. However, its exact implementation is dependent on the platform on which the binary is deployed. Finally, using the same custom Wasm LLVM backend as CROW, we generate a standalone multivariant Wasm binary. Once generated, the multivariant Wasm binary can be deployed to any Wasm engine.

## 3.2.1 Multivariant call graph

The key component of MEWE consists of combining the variants into a single binary. The core idea is to introduce one dispatcher function per original function with variants. A dispatcher function is a synthetic function in charge of choosing a variant at random when the original function is called. With the introduction of the dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

**Definition 1** *Multivariant Call Graph (MCG): A multivariant call graph is a call graph* $\langle N, E \rangle$ *where the nodes in* $N$ *represent all the functions in the binary and an edge* $(f_1, f_2) \in E$ *represents a possible invocation of* $f_2$ *by* $f_1$ *[***?** *]. The nodes in* $N$ *have three possible types: a function present in the original program, a generated function variant, or a dispatcher function.*

## 3.2.2 Exemplifying a Multivariant binary

In Figure 3.4, we show the original static call graph for an original program (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The gray nodes represent function variants, the green nodes function dispatchers, and the yellow nodes are the original functions. The directed edges represent the possible calls. The original program includes three functions. MEWE generates 43 variants for the first function, none for the second, and three for the third. MEWE introduces two dispatcher nodes for the first and third functions. Each dispatcher is connected to the corresponding function variants to invoke one variant randomly at runtime.

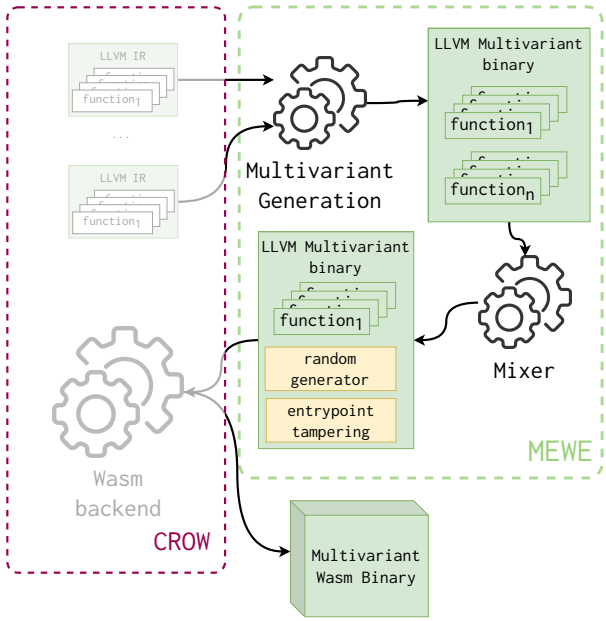



***Figure 3.3:*** *Overview of MEWE workflow. It takes as input an LLVM binary. It first generates a set of functionally equivalent variants for each function in the binary using CROW. Then, MEWE generates an LLVM multivariant binary composed of all the function variants. Finally, the Mixer includes the behavior in charge of selecting a variant when a function is invoked. Finally, the MEWE mixer composes the LLVM multivariant binary with a random number generation library and tampers the original application entrypoint. The final process produces a Wasm multivariant binary ready to be deployed. Figure partially taken from [**?** ].*

In Listing 3.7, we demonstrate how MEWE constructs the function dispatcher, corresponding to the rightmost green node in Figure 3.4, which handles three created variants including the original. The dispatcher function retains the same signature as the original function. Initially, the dispatcher invokes a random number generator, the output of which is used to select a specific function variant for execution (as seen on line 6 in Listing 3.7). To enhance security, we employ a switch-case structure within the dispatcher, mitigating vulnerabilities associated with speculative execution-based attacks [**?** ] (refer to lines 12 to 19 in Listing 3.7). This approach also eliminates the need for multiple function definitions with identical signatures, thereby reducing the potential attack surface in cases where the function signature itself is vulnerable [**?** ]. Additionally, MEWE can inline function variants directly into the dispatcher, obviating the need for redundant definitions (as illustrated on line 16 in Listing 3.7). Remarkably, we prioritize security over performance, i.e., while using indirect

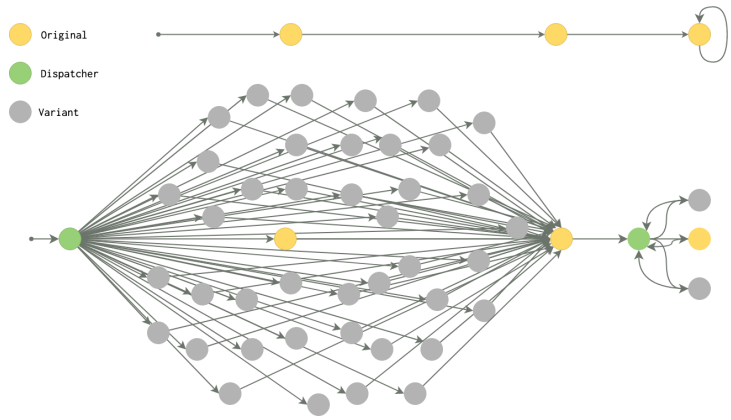



***Figure 3.4:*** *Example of two static call graphs. At the top, is the original call graph, and at the bottom, is the multivariant call graph, which includes nodes that represent function variants (in gray), dispatchers (in green), and original functions (in yellow). Figure taken from [**?** ].*

calls in place of a switch-case could offer constant-time performance benefits, we implement switch-case structures.

```
2  ; Multivariant foo wrapping ;
3  define internal i32 @foo(i32 %0) {
4      entry:
5        ; It first calls the dispatcher to discriminate between the created
             variants ;
6        %1 = call i32 @discriminate(i32 3)
7        switch i32 %1, label %end [
8          i32 0, label %case_43_
9          i32 1, label %case_44_
10       ]
11     ;One case for each generated variant of foo ;
12     case_43_:
13       %2 = call i32 @foo_43_(%0)
14       ret i32 %2
15     case_44_:
16       ; MEWE can inline the body of the a function variant ;
17       %3 = <body of foo_44_ inlined>
18       ret i32 %3
19     end:
20       ; The original is also included ;
21       %4 = call i32 @foo_original(%0)
22       ret i32 %4
23 }
```

***Listing 3.7:*** *Dispatcher function embedded in the multivariant binary of the original function in the rightmost green node in Figure 3.4. The code is commented for the sake of understanding.*

In Listing 3.7, we illustrate the LLVM construction for the function dispatcher corresponding to the right most green node of Figure 3.4. Notice that, the dispatcher function is constructed using the same signature as the original function. It first calls the random generator, which returns a value used to invoke a specific function variant (see line 6 in Listing 3.7). We utilize a switch-case structure in the dispatchers to prevent indirect calls, which are vulnerable to speculative execution-based attacks [**?** ] (see lines 12 to 19 in Listing 3.7), i.e., the choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [**?** ]. In addition, MEWE can inline function variants inside the dispatcher instead of defining them again (see line 16 in Listing 3.7). Remarkably, we trade security over performance since dispatcher functions that perform indirect calls, instead of a switch-case, could improve the performance of the dispatchers as indirect calls have constant time.

**Contribution paper and artifact**

MEWE provides dynamic execution path randomization by packaging variants generated out of CROW.
MEWE is fully presented in Cabrera-Arteaga et al. "Multi-Variant Execution at the Edge" *Proceedings of Moving Target Defense, 2022, ACM* `https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

MEWE is also available as an open-source tool at `https://github.com/ASSERT-KTH/MEWE`

## 3.3 WASM-MUTATE: Fast and Effective Binary for WebAssembly

In this section, we introduce our third technical contribution, WASM-MUTATE [**?** ], a tool that generates thousands of functionally equivalent variants out from a WebAssembly binary input. Leveraging rewriting rules and e-graphs [**?** ] for software diversification, WASM-MUTATE synthesizes program variants by transforming parts of the original binary. In Figure 3.1, we highlight WASM-MUTATE as the blue squared tooling.

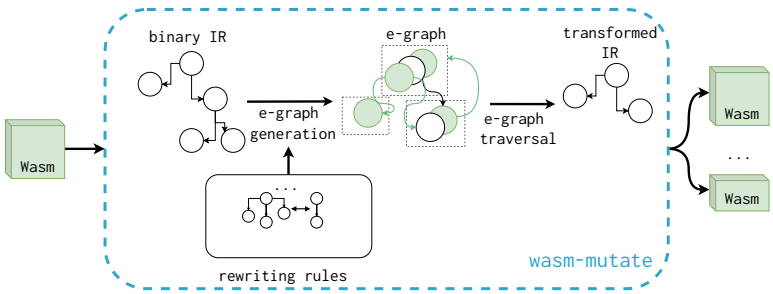



***Figure 3.5:*** *WASM-MUTATE high-level architecture. It generates functionally equivalent variants from a given WebAssembly binary input. Its central approach involves synthesizing these variants by substituting parts of the original binary using rewriting rules, boosted by diversification space traversals using e-graphs.*

Figure 3.5 illustrates the workflow of WASM-MUTATE, which initiates with a WebAssembly binary as its input. The first step involves parsing this binary to create suitable abstractions, e.g. an intermediate representation. Subsequently, WASM-MUTATE utilizes predefined rewriting rules to construct an e-graph for the initial program, encapsulating all potential equivalent codes derived from the rewriting rules. The assurance of functional equivalence is rooted in the inherent

properties of the individual rewrite rules employed. Then, pieces of the original program are randomly substituted by the result of random e-graph traversals, resulting in a variant that maintains functional equivalence to the original binary.

WASM-MUTATE applies one transformation at a time. Notice that, the output of one applied transformation can be chained again as an input WebAssembly binary, enabling the generation of many variants, leading us to enunciate the notion of *Stacked transformation*

**Definition 2** *Stacked transformation: Given an original input WebAssembly binary $I$ and a diversifier $D$, stacked transformations are defined as the application of $D$ over the binary $I$ multiple times, i.e., $D(D(D(...(I))))$. Notice that, the number of stacked transformations are the number of times the diversifier $D$ is applied.*

### 3.3.1 WebAssembly Rewriting Rules

WASM-MUTATE contains a comprehensive set of 135 rewriting rules. In this context, a rewriting rule is a tuple `(LHS, RHS, Cond)` where `LHS` specifies the segment of binary targeted for replacement, `RHS` describes its functionally equivalent substitute, and `Cond` outlines the conditions that must be met for the replacement to take place, e.g. enhancing type constraints. WASM-MUTATE groups these rewriting rules into meta-rules depending on their target inside a Wasm binary, ranging from high-level changes affecting binary section structure to low-level modifications within the code section. This section focuses on the biggest meta-rule implemented in WASM-MUTATE, the `Peephole` meta-rule [3].

Rewriting rules inside the *Peephole* meta-rule, operate over the data flow graph of instructions within a function body, representing the lowest level of rewriting. In WASM-MUTATE, we have implemented 125 rewriting rules specifically for this category, each one avoiding targeting instructions that might induce undefined behavior, e.g., function calls.

Moreover, we augment the internal representation of a Wasm program to bolster WASM-MUTATE's transformation capabilities through the `Peephole` meta-rule. Concretely, we augment the parsing stage in WASM-MUTATE by including custom operator instructions. These custom operator instructions are designed to use well-established code diversification techniques through rewriting rules. When converting back to the WebAssembly binary format from the intermediate representation, custom instructions are meticulously handled to retain the original functionality of the WebAssembly program.

In the following example, we demonstrate a rewriting rule within the `Peephole` meta-rule that utilizes a custom `rand` operator to expand statically declared constants within any WebAssembly program function body. The unfold rewriting rule, as the name suggests, transforms statically declared constants into the sum

[3] For an in-depth explanation of the remaining meta-rules, refer to [**?** ].

of two random numbers. During the generation of the WebAssembly variant, the custom `rand` operator is substituted with a randomly chosen static constant. Notice that the condition specified in the last part of the rewriting rule ensures that this predicate is satisfied.

```
LHS i32.const x

---------------------------------------------------------------

RHS (i32.add (i32.rand i32.const y))

Cond y = x - i32.rand
```

Although this rewriting approach may appear simplistic, especially because compilers often eliminate it through *Constant Folding* optimization [**?** ], it stresses on the spill/reload component of the compiler when the WebAssembly binary is JITed to machine code. Spill/reloads occur when the compiler runs out of physical registers to store intermediate calculations, resorting to specific memory locations for temporary storage. The unfold rewriting rule indirectly stresses this segment of memory. Notably, with this specific rewriting rule, we have found a CVE in the wasmtime standalone engine [**?** ].

### 3.3.2 E-Graphs traversals

We developed WASM-MUTATE leveraging e-graphs, a specific graph data structure for representing and applying rewriting rules [**?** ]. In the context of WASM-MUTATE, e-graphs are constructed from the input WebAssembly program and the implemented rewriting rules (we detail the e-graph construction process in Section 3 of [**?** ]).

Willsey et al. highlight the potential for high flexibility in extracting code fragments from e-graphs, a process that can be recursively orchestrated through a cost function applied to e-nodes and their respective operands. This methodology ensures the functional equivalence of the derived code [**?** ]. For instance, e-graphs solve the problem of providing the best code out of several optimization rules [**?** ]. To extract the "optimal" code from an e-graph, one might commence the extraction at a specific e-node, subsequently selecting the AST with the minimal size from the available options within the corresponding e-class's operands. In omitting the cost function from the extraction strategy leads us to a significant property: *any path navigated through the e-graph yields a functionally equivalent code variant.*

We exploit such property to fastly generate diverse WebAssembly variants. We propose and implement an algorithm that facilitates the random traversal of an e-graph to yield functionally equivalent program variants, as detailed in Algorithm 1. This algorithm operates by taking an e-graph, an e-class node (starting with the root's e-class), and a parameter specifying the maximum extraction depth of the expression, to prevent infinite recursion. Within the

algorithm, a random e-node is chosen from the e-class (as seen in lines 5 and 6), setting the stage for a recursive continuation with the offspring of the selected e-node (refer to line 8). Once the depth parameter reaches zero, the algorithm extracts the most concise expression available within the current e-class (line 3). Following this, the subexpressions are built (line 10) for each child node, culminating in the return of the complete expression (line 11).

**Algorithm 1** e-graph traversal algorithm taken from [**?** ].

```
1:  procedure TRAVERSE(egraph, eclass, depth)
2:      if depth = 0 then
3:          return smallest_tree_from(egraph, eclass)
4:      else
5:          nodes ← egraph[eclass]
6:          node ← random_choice(nodes)
7:          expr ← (node, operands = [])
8:          for each child ∈ node.children do
9:              subexpr ← TRAVERSE(egraph, child, depth − 1)
10:             expr.operands ← expr.operands ∪ {subexpr}
11:         return expr
```

### 3.3.3 Exemplifying WASM-MUTATE

Let us illustrate how WASM-MUTATE generates variant programs by using the before enunciated algorithm. Here, we use Algorithm 1 with a maximum depth of 1. In Listing 3.8 a hypothetical original Wasm binary is illustrated. In this context, a potential user has set two pivotal rewriting rules: `(x, container (x nop),)` and `(x, x i32.add 0, x instanceof i32)`. The former rule, grants the ability to append a `nop` instruction to any subexpression, a well-known low-level diversification strategy [**?** ]. Conversely, the latter rule adds zero to any numeric value.

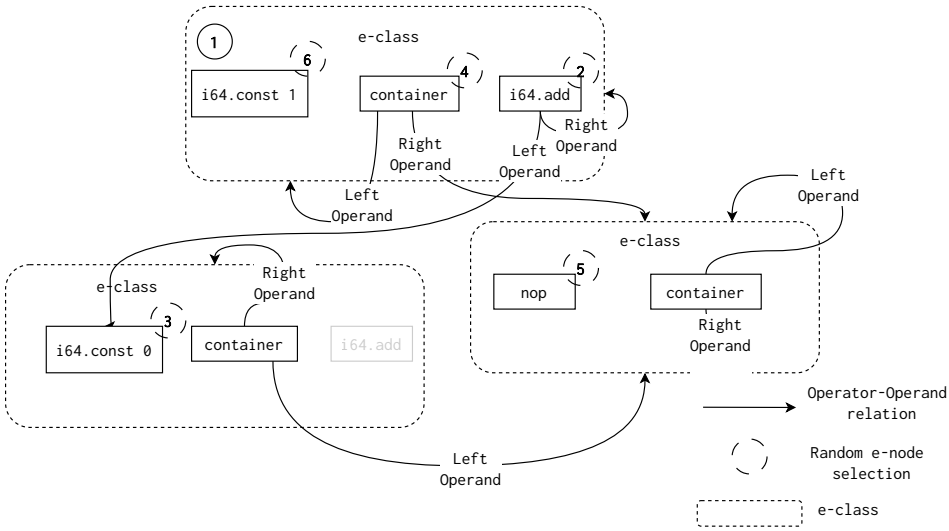



***Figure 3.6:*** *e-graph built for rewriting the first instruction of Listing 3.8.*

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        i64.const 1)
)
```

***Listing 3.8:*** *Wasm function.*

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        (i64.add (
            i64.const 0
            i64.const 1
            nop
        ))
    )
```

***Listing 3.9:*** *Random peephole mutation using egraph traversal for Listing 3.8 over e-graph Figure 3.6. The textual format is folded for better understanding.*

Leveraging the code presented in Listing 3.8 alongside the defined rewriting rules, we build the e-graph, simplified in Figure 3.6. In the figure, we highlight various stages of Algorithm 1 in the context of the scenario previously described. The algorithm initiates at the e-class with the instruction `i64.const 1`, as seen in Listing 3.8. At ②, it randomly selects an equivalent node within the e-class, in this instance taking the `i64.add` node, resulting: `expr`

`= i64.add l r`. As the traversal advances, it follows on the left operand of the previously chosen node, settling on the `i64.const 0` node within the same e-class ③. Then, the right operand of the `i64.add` node is choosen, selecting the `container` ④ operator yielding: `expr = i64.or (i64.const 0 container ( r nop ))`. The algorithm chooses the right operand of the `container` ⑤, which correlates to the initial instruction e-node highlighted in ⑥, culminating in the final expression: `expr = i64.or (i64.const 0 container( i64.const 1 nop)) i64.const 1`. As we proceed to the encoding phases, the `container` operator is ignored as a real Wasm instruction, finally resulting in the program in Listing 3.9.

Notice that, within the e-graph showcased in Figure 3.6, the container node maintains equivalence across all e-classes. Consequently, increasing the depth parameter in Algorithm 1 would potentially escalate the number of viable variants infinitely.

**Contribution paper and artifact**

WASM-MUTATE uses hand-made rewriting rules and random traversals over e-graphs to provide a binary-based solution for WebAssembly diversification.
WASM-MUTATE is fully presented in Cabrera-Arteaga et al. "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly" *Under review at Computers & Security* `https://arxiv.org/pdf/2309.07638.pdf`.

WASM-MUTATE is available at `https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-mutate` as a contribution to the Bytecode Alliance organization [a]. The Bytecode Alliance is dedicated to creating secure new software foundations, building on standards such as WebAssembly and WASI.

[a] `https://bytecodealliance.org/`

## 3.4 Comparing CROW, MEWE, and WASM-MUTATE

In this section, we compare CROW, MEWE, and WASM-MUTATE, highlighting their key differences. These distinctions are summarized in Table 3.1. The table is organized into columns that represent attributes of each tool: the tool's name, input format, core diversification strategy, number of variants generated within an hour, targeted sections of the WebAssembly binary for diversification, strength of the generated variants, and the security applications of these variants. Each row in the table corresponds to a specific tool. The *Variant strength* accounts for the capability of each tool on generating variants that are preserved after the JIT

compilation of V8 and wasmtime in average. For example, a higher value of the *Variant strength* indicates that the generated variants are not reversed by JIT compilers, ensuring that the diversification is preserved in an end-to-end scenario of a WebAssembly program, i.e. from the source code to its final execution. Notice that, the data and insights presented in the table are sourced from the respective papers of each tool and, from the previous discussion in this chapter.

CROW is a compiler-based strategy, needing access to the source code or its LLVM IR representation to work. Its core is an enumerative synthesis implementation with functionallity verification using SMT solvers, ensuring the functional equivalence of the generated variants. In addition, MEWE extends the capabilities of CROW, utilizing the same underlying technology to create program variants. It goes a step further by packaging the LLVM IR variants into a WebAssembly multivariant, providing MVE through execution path randomization. Both CROW and MEWE are fully automated, requiring no user intervention besides the input source code. WASM-MUTATE, on the other hand, is a binary-based tool. It uses a set of rewriting rules and the input Wasm binary to generate program variants, centralizing its core around random e-graph traversals. Remarkably, WASM-MUTATE removes the need for compiler adjustments, offering compatibility with any existing WebAssembly binary.

We have observed several interesting phenomena when aggregating the empirical data presented in the corresponding papers of CROW, MEWE and WASM-MUTATE [**? ? ?** ]. This can be appreciated in the fourth, fifth and sixth columns of Table 3.1. We have observed that WASM-MUTATE generates more unique variants in one hour than CROW and MEWE in at least one order of magnitude. This is mainly because of three reasons. First, CROW and MEWE rely on SMT solvers to prove functionally equivalence, placing a bottleneck when generating variants. Second, CROW and MEWE generation capabilities are limited by the *overlapping* phenomenon discussed in Section 3.1.3. Third, WASM-MUTATE can generate variants in any part of the Wasm binary, while CROW and MEWE are limited to the code and function sections.

On the other hand, CROW and MEWE, by using enumerative synthesis, ensure that the generated variants are more preserved than the variants created by WASM-MUTATE. In other words, the transformations generated out of CROW and MEWE are virtually irreversible by JIT compilers, such as V8 and wasmtime. This phenomenon is highlighted in the *Variants strength* column of Table 3.1, where we show that CROW and MEWE generate variants with 96% of preservation against 75% of WASM-MUTATE. High preservation is especially important where the preservation of the diversification is crucial, e.g. to hinder reverse engineering.

| **Tool** | **Input** | **Core** | **Variants in 1h** | **Target** | **Variants Strength** | **Security applications** |
|---|---|---|---|---|---|---|
| CROW | Source code or LLVM Ir | Enumerative synthesis with functional equivalence proved through SMT solvers | > 1k | Code section | **96%** | Hinders static analysis and reverse engineering. |
| MEWE | Source code or LL VM Ir | CROW, Multivariant execution | > 1k | Code and Function sections | **96%** | Hinders static and dynamic analysis, reverse engineering and, web timing-based attacks. |
| WASM-MUTATE | **Wasm binary** | hand-made rewriting rules, e-graph random traversals | **> 10k** | **All WebAssembly sections** | 76% | Hinders signature-based identification, and cache timing side-channel attacks. |

***Table 3.1:*** *Comparing CROW, MEWE and WASM-MUTATE. The table columns are: the tool's name, input format, core diversification strategy, number of variants generated within an hour, targeted sections of the WebAssembly binary, strength of the generated variants, and the security applications of these variants. The* Variant strength *accounts for the capability of each tool on generating variants that are preserved after the JIT compilation of V8 and wasmtime in average. Our three technical contributions are complementary tools that can be combined.*

**Takeaway**

Our three technical contributions serve as complementary tools that can be combined. For instance, when the source code for a WebAssembly binary is either non-existent or inaccessible, WASM-MUTATE offers a viable solution for generating code variants. On the other hand, CROW and MEWE excel in scenarios where high preservation is crucial.

### 3.4.1 Security applications

The final column of Table 3.1 emphasizes the security benefits derived from the variants produced by our three key technical contributions. One immediate advantage of altering the structure of WebAssembly binaries across different variants is the mitigation of signature-based identification, thereby enhancing resistance to static reverse engineering. Additionally, our tools generate a diverse array of code variants that are highly preserved. This implies that these variants, each with their unique WebAssembly code, retain their distinct characteristics even after being translated into machine code by JIT compilers. This high level of preservation significantly mitigates the risks associated with side-channel attacks that target specific machine code instructions, such as port contention attacks [**?** ]. For instance, if a WebAssembly binary is transformed in such a manner that its resulting machine code instructions differ from the original, it becomes more challenging for a side-channel attack. Conversely, if the compiler translates the variant into machine code that closely resembles the original, the side-channel attack could still exploit those instructions to extract information about the original WebAssembly binary.

Altering the layout of a WebAssembly program inherently influences its managed memory during runtime (see Section 2.1.4). This phenomenon is especially important for CROW and MEWE, given that they do not directly address the WebAssembly memory model. Significantly, CROW and MEWE considerably alter the managed memory by modifying the layout of the WebAssembly program. For example, the *constant inferring* transformations significantly alter the layout of program variants, affecting unmanaged memory elements such as the returning address of a function. Furthermore, WASM-MUTATE not only affects managed memory through changes in the WebAssembly program layout. It also adds rewriting rules to transform unmanaged memory instructions. Memory alterations, either to the unmanaged or managed memories, have substantial security implications, by eliminating potential cache timing side-channels [**?** ].

Last but not least, our technical contributions enhance security against web timing-based attacks [**?** ] by creating variants that exhibit a wide range of execution times, including faster variants compared to the original program.

This strategy is especially prominent in MEWE's approach, which develops multivariants functioning on randomizing execution paths, thereby thwarting attempts at timing-based inference attacks [**?** ]. Adding another layer benefit from MEWE, the integration of diverse variants into multivariants can potentially disrupt dynamic reverse engineering tools such as symbolic executors [**?** ]. Concretely, different control flows through a random discriminator, exponentially increase the number of possible execution paths, making multivariant binaries virtually unexplorable.

**Takeaway**

CROW, MEWE and WASM-MUTATE generate WebAssembly variants that can be used to enhance security. Overall, they generate variants that are suitable for hardening static and dynamic analysis, side-channel attacks, and, to thwart signature-based identification.

## ■ Conclusions

In this chapter, we discuss the technical specifics underlying our primary technical contributions. We elucidate the mechanisms through which CROW generates program variants. Subsequently, we discuss MEWE, offering a detailed examination of its role in forging MVE for WebAssembly. We also explore the details of WASM-MUTATE, proposing a novel e-graph traversal algorithm to fast spawn Wasm program variants. Remarkably, we undertake a comparative analysis of the three tools, highlighting their respective benefits and limitations, alongside the potential security applications of the generated Wasm variants.

In Chapter 4, we present two use cases that support the exploitation of these tools. Chapter 4 serves to bridge theory with practice, showcasing the tangible impacts and benefits realized through the deployment of CROW, MEWE, and WASM-MUTATE.

# 4
# EXPLOITING SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

*If you find that you're spending all your time on theory, start turning some attention to practical things; it will improve your theories. If you find that you're spending almost all your time on practice, start turning some attention to theoretical things; it will improve your practice.*

— Donald Knuth

In this chapter, we illustrate the application of Software Diversification for both offensive and defensive purposes. We discuss two selected use cases that demonstrate practical applications of our contributions. Additionally, we discuss the challenges and benefits arising from the application of Software Diversification to WebAssembly.

## 4.1 Offensive Diversification: Malware evasion

The primary malicious use of WebAssembly in browsers is cryptojacking [**?** ]. This is due to the essence of cryptojacking, the faster the mining, the better. Let us illustrate how a malicious WebAssembly binary is involved into browser cryptojacking. Figure 4.1 illustrates a browser attack scenario: a practical WebAssembly cryptojacking attack consists of three components: a WebAssembly binary, a JavaScript wrapper, and a backend cryptominer pool. The WebAssembly binary is responsible for executing the hash calculations, which consume significant computational resources. The JavaScript wrapper facilitates the communication between the WebAssembly binary and the cryptominer pool.

[0] Compilation probe time 2023/11/07 12:55:02

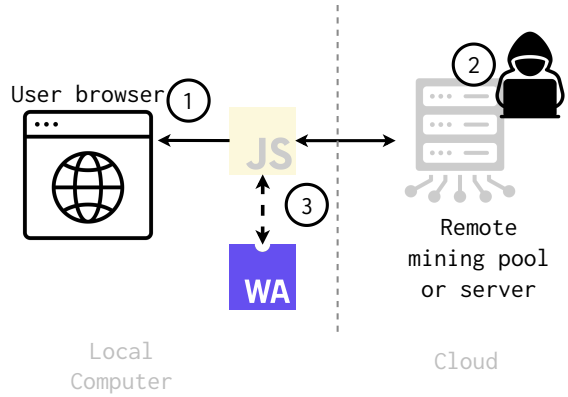



***Figure 4.1:*** *A remote mining pool server, a JavaScript wrapper and the WebAssembly binary form the triad of a cryptojacking attack in browser clients.*

The aforementioned components require the following steps to succeed in cryptomining. First, the victim visits a web page infected with the cryptojacking code. The web page establishes a channel to the cryptominer pool, which then assigns a hashing job to the infected browser. The WebAssembly cryptominer calculates thousands of hashes inside the browser. Once the malware server receives acceptable hashes, it is rewarded with cryptocurrencies for the mining. Then, the server assigns a new job, and the mining process starts over.

Both antivirus software and browsers have implemented measures to detect cryptojacking. For instance, Firefox employs deny lists to detect cryptomining activities [**?** ]. The academic community has also contributed to the body of work on detecting or preventing WebAssembly-based cryptojacking, as outlined in Section 2.1.6. However, malicious actors can employ evasion techniques to circumvent these detection mechanisms. Bhansali et al. are among the first who have investigated how WebAssembly cryptojacking could potentially evade detection [**?** ], highlighting the critical importance of this use case. The case illustrated in the subsequent sections uses Offensive Software Diversification for evading malware detection in WebAssembly.

### 4.1.1 Cryptojacking defense evasion

Considering the previous scenario, several techniques can be directly implemented in browsers to thwart cryptojacking by identifying the malicious WebAssembly components. Such defense scenario is illustrated in Figure 4.2, where the WebAssembly malicious binary is blocked in ③. The primary aim of our use case is to investigate the effectiveness of code diversification as a means to circumvent cryptojacking defenses. Specifically, we assess whether the following evasion workflow can successfully bypass existing security measures:

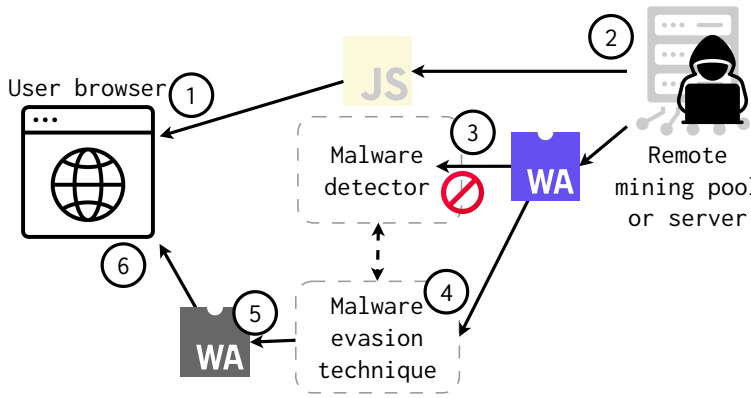



***Figure 4.2:*** *Cryptojacking scenario in which the malware detection mechanism is bypassed by using an evasion technique.*

1. The user loads a webpage infected with cryptojacking malware, which leverages network resources for execution—corresponding to ① and ② in Figure 4.2.

2. A malware detection mechanism (malware oracle) identifies and blocks malicious WebAssembly binaries at ③. For example, a network proxy could intercept and forward these resources to an external detection service via its API.

3. Anticipating that a specific malware detection system is consistently used for defense, the attacker swiftly generates a variant of the WebAssembly cryptojacking malware designed to evade detection at ④.

4. The attacker delivers the modified binary instead of the original one ⑤, which initiates the cryptojacking process and compromises the browser ⑥. The detection method is not capable of detecting the malicious nature of the binary, and the attack is successful.

## 4.1.2 Methodology

Our aim is to empirically validate the workflow in Figure 4.2, i.e., using Offensive Software Diversification in evading malware detection systems. To achieve this, we employ WASM-MUTATE for generating WebAssembly malware variants. In this study, we categorize malware detection mechanisms as malware oracles, which can be of two types: binary and numeric. A binary oracle provides a binary decision, labeling a WebAssembly binary as either malicious or benign. In contrast, a numeric oracle returns a numerical value representing the confidence level of the detection.

**Definition 3** *Malware oracle: A malware oracle is a detection mechanism that returns either a binary decision or a numerical value indicating the confidence level of the detection.*

We employ VirusTotal as a numeric oracle and MINOS [**?** ] as a binary oracle. VirusTotal is an online service that analyzes files and returns a confidence score in the form of the number of antivirus that flag the input file as malware, thus qualifying as a numeric oracle. MINOS, on the other hand, converts WebAssembly binaries into grayscale images and employs a convolutional neural network for classification. It returns a binary decision, making it a binary oracle.

We use the wasmbench dataset [**?** ] to establish a ground truth. After running the wasmbench dataset through VirusTotal and MINOS, we identify 33 binaries that are: 1) flagged as malicious by at least one VirusTotal vendor and, 2) are also detected by MINOS. Then, to simulate the evasion scenario in Figure 4.2, we use WASM-MUTATE to generate WebAssembly binary variants to evade malware detection ( ④ in Figure 4.2). We use WASM-MUTATE in two configurations: feedback-guided and stochastic diversification.

**Definition 4** *Feedback-guided Diversification: In feedback-guided diversification, the transformation process of a WebAssembly program is guided by a numeric oracle, which influences the probability of each transformation. For instance, WASM-MUTATE can be configured to apply transformations that minimize the oracle's confidence score. Note that feedback-guided diversification needs a numeric oracle.*

**Definition 5** *Stochastic Diversification: Unlike feedback-guided diversification, in stochastic diversification, each transformation has an equal likelihood of being applied to the input WebAssembly binary.*

Based on the two types of malware oracles and diversification configurations, we examine three scenarios: 1) VirusTotal with a feedback-guided diversification, 2) VirusTotal with an stochastic diversification, and 3) MINOS with a stochastic diversification. Notice that, the fourth scenario with MINOS and a feedback-guided diversification is not feasible, as MINOS is a binary oracle and cannot provide the numerical values required for feedback-guided diversification.

Our evaluation focuses on two key metrics: the success rate of evading detection mechanisms in VirusTotal and MINOS across the 33 flagged binaries, and the correctness of the generated variants.

**Definition 6** *Evasion rate: This measures the efficacy of WASM-MUTATE in bypassing malware detection systems. For each flagged binary, we input it into WASM-MUTATE, configured with the selected oracle and diversification strategy. We then iteratively apply transformations to the output from the preceding step. This iterative process is halted either when the binary is no longer flagged by the oracle or when a maximum of 1000 stacked transformations have been applied (see Definition 2). This process is repeated with 10 random seeds per binary to*

*simulate 10 different evasion experiments per binary.*

**Definition 7** *Correctness: This verifies the functional equivalence of the variants generated by WASM-MUTATE compared to the original binary. We execute the variants that entirely evade VirusTotal, using controlled and stochastic diversification configurations with WASM-MUTATE for both metrics. Our selection is limited to variants that allow us to fully reproduce the three components displayed in Figure 4.1. We then gather the hashes generated by the cryptojacking binaries and their generation speed, comparing these hashes with those from the original binary. If the hashes match, and the variant executes without error, with the minerpool component validating the hash, we can consider the variant as functionally equivalent.*

### 4.1.3 Results

In Table 4.1, we present a comprehensive summary of the evasion experiments presented in [**?** ], focusing on two oracles: VirusTotal and MINOS[**?** ]. The table is organized into two main categories to separate the results for each malware oracle. For VirusTotal, we further subdivide the results based on the two diversification configurations we employ: stochastic and feedback-guided diversification. In these subsections, the columns indicate the number of VirusTotal vendors that flag the original binary as malware (#D), the maximum number of successfully evaded detectors (Max. #evaded), and the average number of transformations required (Mean #trans.) for each sample. We highlight in bold text the values for which the stochastic diversification or feedback-guided diversification setups best, the lower, the better. The MINOS section solely includes a column that specifies the number of transformations needed for complete evasion. The table has 33 + 1 rows, each representing a unique WebAssembly malware study subject. The final row offers the median number of transformations required for evasion across our evaluated setups and oracles.

**Stochastic diversification to evade VirusTotal:** We execute a stochastic diversification with WASM-MUTATE, setting a limit of 1000 iterations for each binary. In every iteration, we query VirusTotal to determine if the newly generated binary can elude detection. We repeat this procedure with ten distinct seeds for each binary, replicating ten different evasion experiments. As the stochastic diversification section of Table 4.1 illustrates, we successfully produce variants that fully evade detection for 30 out of 33 binaries. The average amount of iterations required to produce a variant that evades all detectors oscillates between 120 to 635 stacked transformations. The mean number of iterations needed never exceeds 1000 stacked transformations. However, three binaries remain detectable under the stochastic diversification setup. In these instances, the algorithm fails to evade 5 out of 31, 6 out of 30, and 5 out of 26 detectors. This shortfall can be attributed to the maximum number of iterations, 1000,

| | VirusTotal | | | | | MINOS[**?** ] |
|---|---|---|---|---|---|---|
| Hash | #D | Stochastic diversification | | Feedback-guided diversification | | |
| | | Max. evaded | Mean trans. | Max. evaded | Mean trans. | Mean trans. |
| 47d29959 | 31 | **26** | N/A | 19 | N/A | 100 |
| 9d30e7f0 | 30 | **24** | N/A | 17 | N/A | 419 |
| 8ebf4e44 | 26 | **21** | N/A | 13 | N/A | 92 |
| c11d82d | 20 | 20 | **355** | 20 | 446 | 115 |
| 0d996462 | 19 | 19 | **401** | 19 | 697 | 24 |
| a32a6f4b | 18 | 18 | 635 | 18 | **625** | 1 |
| fbdd1efa | 18 | 18 | **310** | 18 | 726 | 1 |
| d2141ff2 | 9 | 9 | **461** | 9 | 781 | 81 |
| aafff587 | 6 | 6 | 484 | 6 | **331** | 1 |
| 046dc081 | 6 | 6 | 404 | 6 | **159** | 33 |
| 643116ff | 6 | 6 | **144** | 6 | 436 | 47 |
| 15b86a25 | 4 | 4 | 253 | 4 | **131** | 1 |
| 006b2fb6 | 4 | 4 | **282** | 4 | 380 | 1 |
| 942be4f7 | 4 | 4 | 200 | 4 | 200 | 29 |
| 7c36f462 | 4 | 4 | 236 | 4 | **221** | 85 |
| fb15929f | 4 | 4 | **297** | 4 | 475 | 1 |
| 24aae13a | 4 | 4 | **252** | 4 | 401 | 980 |
| 000415b2 | 3 | 3 | 302 | 3 | **34** | 960 |
| 4cbdbbb1 | 3 | 3 | 295 | 3 | **72** | 1 |
| 65debcbe | 2 | 2 | 131 | 2 | **33** | 38 |
| 59955b4c | 2 | 2 | 130 | 2 | **33** | 38 |
| 89a3645c | 2 | 2 | 431 | 2 | **107** | 108 |
| a74a7cb8 | 2 | 2 | 124 | 2 | **33** | 38 |
| 119c53eb | 2 | 2 | 104 | 2 | **18** | 1 |
| 089dd312 | 2 | 2 | 153 | 2 | **123** | 68 |
| c1be4071 | 2 | 2 | 130 | 2 | **33** | 38 |
| dceaf65b | 2 | 2 | 140 | 2 | 132 | 66 |
| 6b8c7899 | 2 | 2 | 143 | 2 | **33** | 38 |
| a27b45ef | 2 | 2 | 145 | 2 | **33** | 33 |
| 68ca7c0e | 2 | 2 | 137 | 2 | **33** | 38 |
| f0b24409 | 2 | 2 | 127 | 2 | **11** | 33 |
| 5bc53343 | 2 | 2 | 118 | 2 | **33** | 33 |
| e09c32c5 | 1 | 1 | **120** | 1 | 488 | 15 |
| Median | | | 218 | | 131 | 38 |

***Table 4.1:*** *The table has two main categories for each malware oracle, corresponding to the two oracles we use: VirusTotal and MINOS. For VirusTotal, divide the results based on the two diversification configurations: stochastic and feedback-guided diversification. We provide columns that indicate the number of VirusTotal vendors that flag the original binary as malware (#D), the maximum number of successfully evaded detectors (Max. #evaded), and the average number of transformations required (Mean #trans.) for each sample. We highlight in bold text the values for which diversification setups are best, the lower, the better. The MINOS section includes a column that specifies the number of transformations needed for complete evasion. The final row offers the median number of transformations required for evasion across our evaluated setups and oracles.*

that we employ in our experiments. Increasing iterations further, however, seems unrealistic. If certain transformations enlarge the binary size, a significantly large binary could become impractical due to bandwidth limitations. In summary, stochastic diversification with WASM-MUTATE markedly reduces the detection rate by VirusTotal antivirus vendors for cryptojacking malware, achieving total evasion in 30 out of 33 (90%) cases within the malware dataset.

**Feedback-guided diversification to evade VirusTotal:** stochastic diversification does not guide the diversification based on the number of evaded detectors, it is purely random, and has some drawbacks. For example, some transformations might suppress other transformations previously applied. We have observed that, by carefully selecting the order and type of transformations applied, it is possible to evade detection systems in fewer iterations. This can be appreciated in the results of the feedback-guided diversification part of Table 4.1. The feedback-guided diversification setup successfully generates variants that totally evade the detection for 30 out of 33 binaries, it is thus as good as the stochastic setup. Remarkably, for 21 binaries out of 30, feedback-guided needs only 40% of the calls the stochastic diversification setup needs, demonstrating larger efficiency.

**Stochastic diversification to evade MINOS:** Relying exclusively on VirusTotal for detection could pose issues, particularly given the existence of specialized solutions for WebAssembly, which differ from the general-purpose vendors within VirusTotal. In Section 2.1.6 we highlight several examples of such solutions. Yet, for its simplicity, we extend this experiment by using MINOS[**?** ], an antivirus specifically designed for WebAssembly. The results of evading MINOS can be seen in the final column of Table 4.1. The bottom row of Table 4.1 highlights that fewer iterations are required to evade MINOS than VirusTotal through WebAssembly diversification, indicating a greater ease in eluding MINOS. The stochastic diversification setup requires a median iteration count of 218 to evade VirusTotal. In contrast, the feedback-guided diversification setup necessitates only 131 iterations. Remarkably, a mere 38 iterations are needed for MINOS. WASM-MUTATE evaded detection for 8 out of 33 binaries in a single iteration. This result implies a vulnerability in the MINOS model to binary diversification.

**WebAssembly variants correctness:** To evaluate the correctness of the malware variants created with WASM-MUTATE, we focused on six binaries that we could build and execute end-to-end, as these had all three components outlined in Figure 4.1. We select only six binaries because the process of building and executing the binaries involves three components: the WebAssembly binary, its JavaScript complement, and the miner pool. These components were not found for the remaining 24 evaded binaries in the study subjects. For the six binaries,

we then replace the original WebAssembly code with variants generated using VirusTotal as the malware oracle and WASM-MUTATE for both controlled and stochastic diversification configurations. We then execute both the original and the generated variants. We assess the correctness of the variants by examining the hashes they generate. Our findings show that all variants generated with WASM-MUTATE are correct, i.e., they generate the correct hashes and execute without error. Additionally, we found that 19% of the generated variants surpassed the original cryptojacking binaries in performance.

**Reflection**

Our experiments conclusively demonstrate that WASM-MUTATE can effectively circumvent malware detection systems. A possible key factor behind this is a misguided perception of resilience. Malware detection is a well-known difficult problem [**?** ]. Yet, prior research on static WebAssembly malware detection has shown an erroneous presumption: the existing of only metadata (WebAssembly custom sections) obfuscation, or the complete absence of obfuscation techniques for WebAssembly [**? ? ? ? ?** ]. As explored in Section 2.2, a software diversification engine can potentially function as an obfuscator. The discussed use case partially demonstrates the assumption of non-existing obfuscators might be incorrect. Consequently, our software diversification tools provide a viable solution for enhancing the accuracy of WebAssembly malware detection systems.

**Contribution paper**

WASM-MUTATE generates correct and performant variants of WebAssembly cryptojacking that successfully evade malware detection. The case discussed in this section is fully detailed in Cabrera-Arteaga et al. "WebAssembly Diversification for Malware Evasion" *at Computers & Security, 2023* `https://www.sciencedirect.com/science/article/pii/S0167404823002067`.

## 4.2 Defensive Diversification: Speculative Side-channel protection

As discussed in Section 2.1, WebAssembly is quickly becoming a cornerstone technology in backend systems. Leading companies like Cloudflare and Fastly are championing the integration of WebAssembly into their edge computing platforms, thereby enabling developers to deploy applications that are both modular and securely sandboxed. These server-side WebAssembly applications are generally architected as isolated, single-responsibility services, a model

referred to as Function-as-a-Service (FaaS) [? ? ]. The operational flow of WebAssembly binaries in FaaS platforms is illustrated in Figure 4.3.

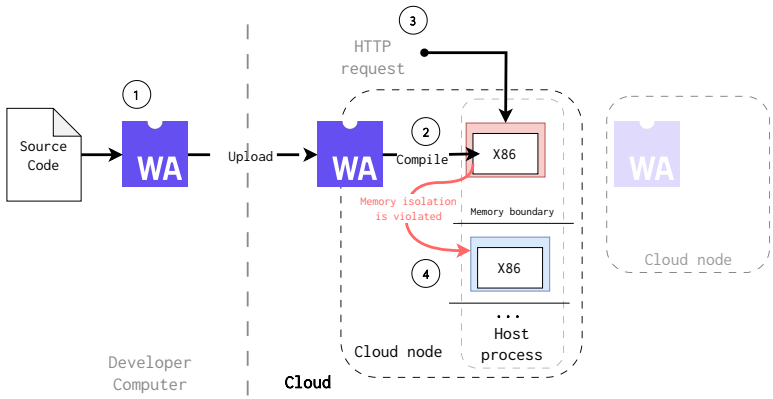



***Figure 4.3:*** *WebAssembly binaries on FaaS platforms. Developers can submit any WebAssembly binary to the platform to be executed as a service in a sandboxed and isolated manner. Yet, WebAssembly binaries are not immune to Spectre attacks.*

The fundamental advantage of using WebAssembly in FaaS platforms lies in its ability to encapsulate thousands of WebAssembly binaries within a singular host process. A developer could compile its source code into a WebAssembly program suitable for the cloud platform and then submit it ( ① in Figure 4.3). This host process is then disseminated across a network of servers and data centers ( ② in Figure 4.3). These platforms convert WebAssembly programs into native code, which is subsequently executed in a sandboxed environment. Host processes can then instantiate new WebAssembly sandboxes for each client function, executing them in response to specific user requests with nanosecond-level latency ( ③ in Figure 4.3). This architecture inherently isolates WebAssembly binary executions from each other as well as from the host process, enhancing security.

However, while WebAssembly is engineered with a strong focus on security and isolation, it is not entirely immune to vulnerabilities such as Spectre attacks [? ? ] ( ④ in Figure 4.3). In the sections that follow, we explore how software diversification techniques can be employed to harden WebAssembly binaries against such attacks.

## 4.2.1 Threat model: speculative side-channel attacks

To illustrate the threat model concerning WebAssembly programs in FaaS platforms, consider the following scenario. Developers, including potentially malicious actors, have the ability to submit any WebAssembly binary to the FaaS platform. A malicious actor could then upload a WebAssembly binary

that, once compiled to native code, employs Spectre attacks. Spectre attacks exploit hardware-based prediction mechanisms to trigger mispredictions, leading to the speculative execution of specific instruction sequences that are not part of the original, sequential execution flow. By taking advantage of this speculative execution, an attacker can potentially access sensitive information stored in the memory allocated to other WebAssembly instance (including itself by violating Control Flow Integrity) or even the host process. Therefore, this poses a significant risk for the overall execution system.

Narayan and colleagues [**?** ] have categorized potential Spectre attacks on WebAssembly binaries into three distinct types, each corresponding to a specific hardware predictor being exploited and a particular FaaS scenario: Branch Target Buffer Attacks, Return Stack Buffer Attacks, and Pattern History Table Attacks defined as follows:

1. The Spectre Branch Target Buffer (btb) attack exploits the branch target buffer by predicting the target of an indirect jump, thereby rerouting speculative control flow to an arbitrary target.
2. The Spectre Return Stack Buffer (rsb) attack exploits the return stack buffer that stores the locations of recently executed call instructions to predict the target of `ret` instructions.
3. The Spectre Pattern History Table (pht) takes advantage of the pattern history table to anticipate the direction of a conditional branch during the ongoing evaluation of a condition.

### 4.2.2 Methodology

Our goal is to empirically validate that Software Diversification can effectively mitigate the risks associated with Spectre attacks in WebAssembly binaries. The green-highlighted section in Figure 4.4 illustrates how Software Diversification can be integrated into the FaaS platform workflow. The core idea is to generate unique and diverse WebAssembly variants that can be randomized at the time of deployment. For this use case, we employ WASM-MUTATE as our tool for Software Diversification.

To empirically demonstrate that Software Diversification can indeed mitigate Spectre vulnerabilities, we reuse the WebAssembly attack scenarios proposed by Narayan and colleagues in their work on Swivel [**?** ]. Swivel is a compiler-based strategy designed to counteract Spectre attacks on WebAssembly binaries by linearizing their control flow during machine code compilation. Our approach differs from theirs in that it is binary-based, compiler-agnostic, and platform-agnostic; we do not propose altering the deployment or toolchain of FaaS platforms.

To measure the efficacy of WASM-MUTATE in mitigating Spectre, we diversify four WebAssembly binaries proposed in the Swivel study. The names

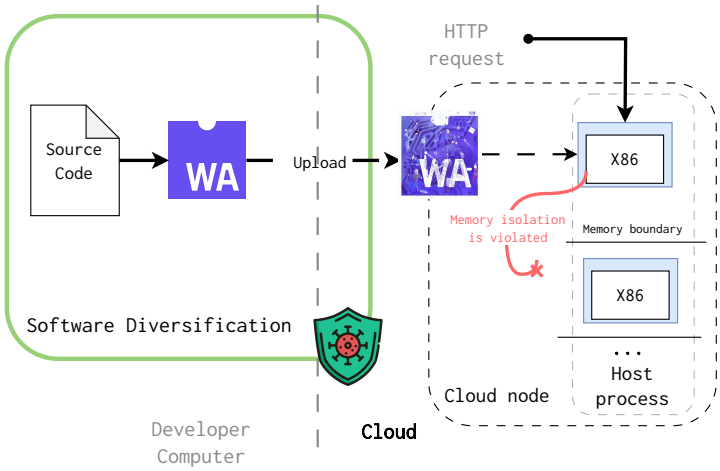



***Figure 4.4:*** *Diversifying WebAssembly binaries to mitigate Spectre attacks in FaaS platforms.*

| Program | Attack |
|---|---|
| btb_breakout | Spectre branch target buffer (btb) |
| btb_leakage | Spectre branch target buffer (btb) |
| ret2spec | Spectre Return Stack Buffer (rsb) |
| pht | Spectre Pattern History Table (pht) |

***Table 4.2:*** *WebAssembly program name and its respective attack.*

of these programs and the specific attacks we examine are available in Table 4.2. For each of these four binaries, we generate up to 1000 random stacked transformations (see Definition 2) using 100 distinct seeds, resulting in a total of 100,000 variants for each original binary. At every 100th stacked transformation for each binary and seed, we assess the impact of diversification on the Spectre attacks by measuring the attack bandwidth for data exfiltration.

**Definition 8** *Attack bandwidth: Given data* $D = \{b_0, b1, ..., b_C\}$ *being exfiltrated in time* $T$ *and* $K = k_0, k_1, ..., k_N$ *the collection of correct data bytes, the bandwidth metric is defined as:*

$$\frac{|b_i \text{ such that } b_i \in K|}{T}$$

The previous metric not only captures the success or failure of the attacks but also quantifies the extent to which data exfiltration is hindered. For example, a variant that still leaks data but does so at an impractically slow rate would be considered hardened against the attack.

### 4.2.3 Results

Figure 4.5 offers a graphical representation of WASM-MUTATE's influence on the Swivel original programs: btb_breakout and btb_leakage with the btb attack. The Y-axis represents the exfiltration bandwidth (see Definition 8). The bandwidth of the original binary under attack is marked as a blue dashed horizontal line. In each plot, the variants are grouped in clusters of 100 stacked transformations. These are indicated by the green violinplots.

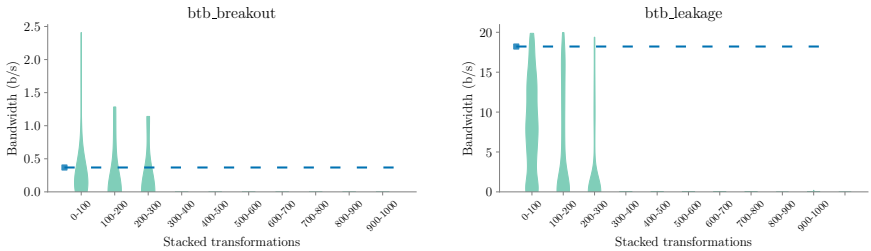



***Figure 4.5:*** *Impact of WASM-MUTATE over btb_breakout and btb_leakage binaries. The Y-axis denotes exfiltration bandwidth, with the original binary's bandwidth under attack highlighted by a blue marker and dashed line. Variants are clustered in groups of 100 stacked transformations, denoted by green violinplots. Overall, for all 100000 variants generated out of each original program, 70% have less data leakage bandwidth. After 200 stacked transformations, the exfiltration bandwidth drops to zero.*

**Population Strength:** For the binaries btb_breakout and btb_leakage, WASM-MUTATE exhibits a high level of effectiveness, generating variants that leak less information than the original in 78% and 70% of instances, respectively. For both programs, after applying 200 stacked transformations, the exfiltration bandwidth drops to zero. This implies that WASM-MUTATE is capable of synthesizing variants that are entirely protected from the original attack. If we consider the results in Table 3.1, generating a variant with 200 stacked transformations can be accomplished in just a matter of seconds for a single WebAssembly binary.

**Effectiveness of WASM-MUTATE:** As illustrated in Figure 4.6, similarly to Figure 4.5, WASM-MUTATE significantly impacts the programs ret2spec and pht when subjected to their respective attacks. In 76% of instances for ret2spec and 71% for pht, the generated variants demonstrated reduced attack bandwidth compared to the original binaries. The plots reveal that a notable decrease in exfiltration bandwidth occurs after applying at least 100 stacked transformations. While both programs show signs of hardening through reduced attack bandwidth, this effect is not immediate and requires a substantial number of transformations

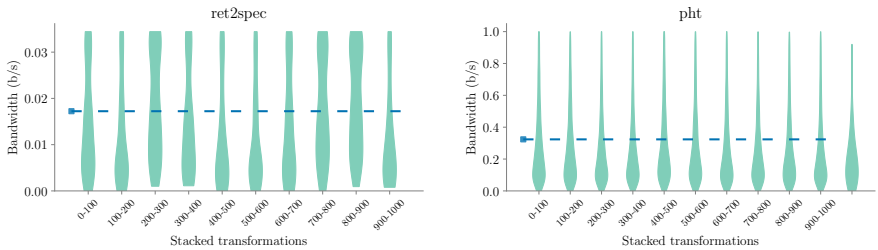



***Figure 4.6:*** *Impact of WASM-MUTATE over ret2spec and pht binaries. The Y-axis denotes exfiltration bandwidth, with the original binary's bandwidth under attack highlighted by a blue marker and dashed line. Variants are clustered in groups of 100 stacked transformations, denoted by green violinplots. Overall, for both programs approximately 70% of the variants have less data leakage bandwidth.*

to become effective. Additionally, the bandwidth distribution is more varied for these two programs compared to the two previous ones. Our analysis suggests a correlation between the reduction in attack bandwidth and the complexity of the binary being diversified. Specifically, ret2spec and pht are substantially larger programs, containing over 300,000 instructions, compared to btb_breakout and btb_leakage, which have fewer than 800 instructions. Therefore, given that WASM-MUTATE performs one transformation per invocation, the probability of affecting critical components to hinder attacks decreases in larger binaries.

**Disrupting timers:** Cache timing side-channel attacks, including for the four binaries analyzed in this use case, depend on precise timers to measure cache access times. Disrupting these timers can effectively neutralize the attack [**?** ]. One key reason our results show variants resilient to Spectre attacks is the approach of WASM-MUTATE. It creates variants that offer a similar approach. Our WebAssembly variants introduce perturbations in the timing steps of WebAssembly variants. This is illustrated in Listing 4.1 and Listing 4.2, where the former shows the original time measurement and the latter presents a variant with introduced operations. By introducing additional instructions, the inherent randomness in the time measurement of a single or a few instructions is amplified, thereby reducing the timer's accuracy.

```
;; Code from original btb_breakout
...
(call $readTimer)
(set_local $end_time)
... access to mem
(i64.sub (get_local $end_time ) (get_local $start_time))
(set_local $duration)
...
```

***Listing 4.1:*** *Wasm timer code.*

```
;; Variant code
...
(call $readTimer)
(set_local $end_time)
<inserted instructions>
... access to mem
<inserted instructions>
(i64.sub (get_local $end_time ) (get_local $start_time))
(set_local $duration)
...
```

***Listing 4.2:*** *WebAssembly variant with more instructions added in between time measurement.*

**Padding speculated instructions:** CPUs have a limit on the number of instructions they can cache. WASM-MUTATE injects instructions to exceed this limit. This effectively disables the speculative execution of memory accesses. This approach is akin to padding [**?** ], as demonstrated in Listing 4.3 and Listing 4.4. This padding disrupts the binary code's layout in memory, hindering the attacker's ability to initiate speculative execution. Even if speculative execution occurs, the memory access does not proceed as the attacker intended.

```
;; Code from original btb_breakout
...
;; train the code to jump here (index 1)
(i32.load (i32.const 2000))
(i32.store (i32.const 83)) ;; just prevent optimization
...
;; transiently jump here
(i32.load (i32.const 339968)) ;; S(83) is the secret
(i32.store (i32.const 83)) ;; just prevent optimization
```

***Listing 4.3:*** *Two jump locations. The top one trains the branch predictor, the bottom one is the expected jump that exfiltrates the memory access.*

```
;; Variant code
...
;; train the code to jump here (index 1)
<inserted instructions>
(i32.load (i32.const 2000))
<inserted instructions>
(i32.store (i32.const 83)) ;; just prevent optimization
...
;; transiently jump here
<inserted instructions>
(i32.load (i32.const 339968)) ;; "S"(83) is the secret
<inserted instructions>
(i32.store (i32.const 83)) ;; just prevent optimization
...
```

***Listing 4.4:*** *WebAssembly variant with more instructions added indindinctly between jump places.*

**TODO** Map with the paper of Forrest. Section 3.3 of Building diverse computer systems.

**Managed memory impact:** The success in diminishing Spectre attacks is mainly explained by the fact that WASM-MUTATE synthesizes variants that effectively alter memory access patterns. We have identified four primary factors responsible for the divergence in memory accesses among WASM-MUTATE generated variants. First, modifications to the binary layout—even those that do not affect executed code—inevitably alter memory accesses within the program's stack. Specifically, WASM-MUTATE generates variants that modify the return addresses of functions, which consequently leads to differences in execution flow and memory accesses. Second, one of our rewriting rules incorporates artificial global values into WebAssembly binaries. The access to these global variables inevitably affects the managed memory (see Section 2.1.4). Third, WASM-MUTATE injects 'phantom' instructions which do not aim to modify the outcome of a transformed function during execution. These intermediate calculations trigger the spill/reload component of the wasmtime compiler, varying spill and reload operations. In the context of limited physical resources, these operations temporarily store values in memory for later retrieval and use, thus creating diverse managed memory accesses (see the example at Section 3.3.1). Finally, certain rewriting rules implemented by WASM-MUTATE replicate fragments of code, e.g., performing commutative operations. These code segments may contain memory accesses, and while neither the memory addresses nor their values change, the frequency of these operations does.

**Reflection**

Beyond Spectre, one can use WASM-MUTATE to mitigate other side-channel attacks. For instance, port contention attacks [**?** ] rely on the execution of specific instructions for a successful attack. Not only WASM-MUTATE, but also our other tools, can alter those instructions, thereby mitigating the attack. The effectiveness of WASM-MUTATE, coupled with its ability to generate numerous variants, establishes it as an apt tool for mitigating side-channel attacks. Consider, for example, applying this on a global FaaS platform scale. In this scenario, one could deploy a unique, hardened variant for each machine and even for every fresh WebAssembly spawned per user request.

**Contribution paper**

WASM-MUTATE crafts WebAssembly binaries that are resilient to Spectre-like attacks. The case discussed in this section is fully detailed in Cabrera-Arteaga et al. "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly" *Under review* `https://arxiv.org/pdf/2309.07638.pdf`.

## Conclusions

In this chapter, we explore Offensive and Defensive Software Diversification applied to WebAssembly. Offensive Software Diversification highlights both the potential and the latent security risks in applying Software Diversification to WebAssembly malware. Our findings suggest potential enhancements to the automatic detection of cryptojacking malware in WebAssembly, e.g., by stressing their resilience with WebAssembly malware variants. Conversely, Defensive Software Diversification serves as a proactive guard, specifically designed to mitigate the risks associated with Spectre attacks.

Moreover, we have conducted experiments with various use cases that are not shown in this chapter. For instance, CROW [**?** ] excels in generating WebAssembly variants that minimize side-channel noise, thereby bolstering defenses against potential side-channel attacks. Alternatively, deploying multivariants from MEWE [**?** ] can thwart high-level timing-based side-channels [**?** ]. Specifically, we conducted experiments on the round-trip times of the generated multivariants and concluded that, at a high level, the timing side-channel information cannot discriminate between variants. In the subsequent chapter, we will summarize the primary conclusions of this dissertation and propose avenues for future research.

# 5 CONCLUSIONS AND FUTURE WORK

*You're bound to be unhappy if you optimize everything.*

— Donald Knuth

Owing to the growing adoption of WebAssembly, we have focused on the security of WebAssembly programs and the potential usages of Software Diversification. This thesis introduces a comprehensive set of methods and tools for Software Diversification in WebAssembly. It includes the technical contributions of this dissertation: CROW, MEWE, and WASM-MUTATE. Additionally, we present specific use cases for exploiting the diversification created for WebAssembly programs. In this chapter, we initially summarize the technical contributions of this dissertation, including an overview of the empirical findings of our research. Finally, we discuss future research directions in WebAssembly Software Diversification.

## 5.1 Summary of technical contributions

Our first tool, CROW, is a compiler-based approach. It uses the LLVM compiler and requires the source code or the LLVM IR representation for its functioning. Its core comprises an enumerative synthesis implementation. CROW ensures the functional equivalence of the generated variants by employing SMT solvers for functionality verification.

MEWE, on the other hand, enhances CROW by using identical core technology to generate program variants. Furthermore, it encapsulates the LLVM IR variants into a WebAssembly multivariant binary, facilitating execution path randomization. Both CROW and MEWE are fully automated systems, necessitating only the input source code from users.

WASM-MUTATE, a binary-based approach, uses a set of rewriting rules and the input Wasm binary to generate program variants. In WASM-MUTATE, the generation of WebAssembly variants primarily involves random e-graph

[0] Compilation probe time 2023/11/07 12:55:02

traversals. Remarkably, WASM-MUTATE eliminates the need for compiler adjustments, thus ensuring compatibility with all existing WebAssembly binaries. Unlike CROW and MEWE, which are confined to code and function sections, WASM-MUTATE can generate variants by transforming any segment of the Wasm binary.

CROW, MEWE, and WASM-MUTATE are open-source, public tools, making their deployment entirely practical. Notably, WASM-MUTATE is currently in use in real-world scenarios to enhance WebAssembly compilers[1].

## 5.2 Summary of empirical findings

According to the comparison of our technical contributions discussed in Chapter 3 and, the results of our use case experiments in Chapter 4 we summarize the following empirical findings.

**Implications of our implementations:** CROW and MEWE depend on SMT solvers to prove functional equivalence in their enumerative synthesis implementation, which can be a bottleneck in variant generation. Consequently, WASM-MUTATE outperforms CROW and MEWE by producing unique variants. It achieves this in at least an order of magnitude greater, within the same timeframe. The main reason is that WASM-MUTATE uses a preset of rewriting rules accompanied by virtually inexpensive random e-graph traversals. The applications of our technical contributions are not orthogonal but complementary. Specifically, one can employ CROW and MEWE to generate a set of variants, which subsequently serve as rewriting rules for WASM-MUTATE. Furthermore, when practitioners require swift generation of variants, they could utilize WASM-MUTATE, accepting a decrease in preservation of the variants.

**Offensive Software Diversification:** We use WASM-MUTATE to illustrate the practical application of Offensive Software Diversification in WebAssembly. Specifically, we employ WASM-MUTATE in generating WebAssembly variants of cryptojacking malware. These variants effectively evade detection from state-of-the-art malware detection systems like VirusTotal and MINOS. Our research verifies the existence of opportunities for the malware detection community to bolster the automatic detection of cryptojacking WebAssembly malware. One potential contributing factor to the success of WASM-MUTATE's evasion is a false sense of resilience. Prior research into the detection of WebAssembly malware has exposed a flawed presumption that obfuscation techniques for WebAssembly are absent [**? ? ? ? ?** ], whilst our software diversification tools present a viable solution for enhancing the precision of WebAssembly malware detection systems.

**Defensive Software Diversification:** Our techniques enhance overall security by facilitating the deployment of unique and diversified WebAssembly binaries,

[1] `https://github.com/bytecodealliance/wasm-tools`

potentially utilizing different variants as needed. For instance, WASM-MUTATE generates Wasm binaries that are resistant to Spectre-like attacks. Given that WASM-MUTATE can generate tens of thousands of variants within minutes, it becomes feasible to deploy a unique variant for each function invocation on FaaS platforms. This rationale applies equally to both CROW and MEWE. Our tools can mitigate other side-channel attacks. For example, CROW also excels at hardening defenses against potential side-channel attacks. In addition, MEWE tackles high-level timing-based side-channels [**?** ].

## 5.3 Future Work

Along with this dissertation we have highlighted several open challenges related to Software Diversification in WebAssembly. These challenges open up several directions for future research. In the following, we outline some of these directions.

**Extending WASM-MUTATE:** WASM-MUTATE may gain advantages from the enumerative synthesis techniques employed by CROW and MEWE. Specifically, WASM-MUTATE could adopt the transformations generated by these tools as rewriting rules. This approach could enhance WASM-MUTATE in two specific ways. First, it could improve the preservation of the variants generated by WASM-MUTATE. Second, this method would inevitably expand the diversification space of WASM-MUTATE e-graphs.

**Program Normalization:** We successfully employed WASM-MUTATE for the evasion of malware detection (see Section 4.1). The proposed mitigation in the prior study involved code normalization as a means of reducing the spectrum of malware variants. Our current work provides insights into the potential effectiveness of this approach. Specifically, a practically costless process of pre-compiling Wasm binaries could be employed as a preparatory measure for malware classifiers. In other words, a Wasm binary can first be JITed to machine code, effectively eliminating malware variants. This approach could substantially enhance the efficiency and precision of malware detection systems.

**Meta-oracles:** Our experiment results in Section 4.1 indicate that VirusTotal surpasses MINOS in detecting WebAssembly cryptojacking. The primary factor contributing to this is VirusTotal's utilization of a broader range of antivirus vendors, which employs various detection strategies. On the other hand, MINOS functions as a binary oracle. This evidence supports the use of multiple malware oracles (meta-oracles) in identifying cryptojacking malware in browsers. In the context of WebAssembly, given the existence of numerous and diverse Wasm-specific detection mechanisms, this strategy is both practical and feasible, yet not explored in the literature.

**Mitigating Port Contention:** Rokicki et al. [**?** ] showcased the potential of a covert side-channel attack using port contention in WebAssembly code

within browsers to violate cross scripting isolation. Side-channels exploiting port contention utilize the competition for shared hardware resources to extract sensitive information from processes. The attacker measures the time taken to access these shared resources to deduce data or behavior of a victim process sharing the same resources. Counteracting these attacks is especially difficult as they leverage fundamental features of the hardware design meant to enhance performance. The success of this attack largely relies on the accurate prediction of Wasm instructions inducing port contention. To tackle this security concern, WebAssembly Software Diversification can be effectively implemented as a browser plugin. Our tools possess the ability to change the WebAssembly instructions acting as port contention predictors with different instructions. This bears a strong resemblance to the impact on timers and padding discussed earlier in Section 4.2. Such a strategy would certainly eradicate the port contention in the particular port utilized for the attack, subsequently hardening browsers against such detrimental activities.

**AI and Software Diversification:** As discussed in Chapter 3, implementing a diversifier at the high language level seems impractical due to the multitude of existing frontends. However, the emergence of Large Language Models (LLMs) and their ability to generate high-level language may address this problem. Nevertheless, we argue that simply connecting the LLM to the diversifier does not provide a complete solution; studies on preservation must also be conducted. Specifically, high-level diversification might lead to low preservation, thereby challenging the assumption of diversification at the low-level. In the context of WebAssembly, considering the wide variety of frontends, utilizing LLMs might be a feasible method for generating Software Diversification. Although preservation poses a problem at a high-level, it could potentially solve the inherited, more challenging issue of transforming programs at the intermediate representation level or WebAssembly bytecode itself.

# REFERENCES


[1] M. R. Cox, *Cinderella: Three hundred and forty-five variants of Cinderella, Catskin, and Cap o'Rushes*. No. 31, Folk-lore Society, 1893.

[2] Tim Berners-Lee, "The WorldWideWeb browser." `https://www.w3.org/People/Berners-Lee/WorldWideWeb.html`, 1990.

[3] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of javascript," in *ECOOP 2010 – Object-Oriented Programming* (T. D'Hondt, ed.), (Berlin, Heidelberg), pp. 126–150, Springer Berlin Heidelberg, 2010.

[4] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, E. Weippl, and F. Wien, "Fast and reliable browser identification with javascript engine fingerprinting," in *Web 2.0 Workshop on Security and Privacy (W2SP)*, vol. 5, p. 4, Citeseer, 2013.

[5] L. Clark, "What makes webassembly fast?," 2017.

[6] D. Yu, A. Chander, N. Islam, and I. Serikov, "Javascript instrumentation for browser security," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, (New York, NY, USA), p. 237–249, Association for Computing Machinery, 2007.

[7] Y. Ko, T. Rezk, and M. Serrano, "Securejs compiler: Portable memory isolation in javascript," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, (New York, NY, USA), p. 1265–1274, Association for Computing Machinery, 2021.

[8] A. Haas, A. Rossberg, D. L. Schuff, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien, "Bringing the web up to speed with webassembly," *PLDI*, 2017.

[9] WebAssembly Community Group, "WebAssembly Specification." `https://webassembly.github.io/spec/core/syntax/index.html`, 2017.

[10] P. Mendki, "Evaluating webassembly enabled serverless approach for edge computing," in *2020 IEEE Cloud Summit*, pp. 161–166, 2020.

[11] M. Jacobsson and J. Wåhslén, "Virtual machine execution for wearables based on webassembly," in *EAI International Conference on Body Area Networks*, pp. 381–389, Springer, Cham, 2018.

[12] Bytecode Alliance , "Bytecode Alliance." `https://bytecodealliance.org/`, 2019.

[13] "Webassembly system interface." `https://github.com/WebAssembly/WASI`, 2021.

[14] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of webassembly," in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020.

[15] Q. Stiévenart, C. De Roover, and M. Ghafari, "Security risks of porting c programs to webassembly," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC '22, (New York, NY, USA), p. 1713–1722, Association for Computing Machinery, 2022.

[16] T. Rokicki, C. Maurice, M. Botvinnik, and Y. Oren, "Port contention goes portable: Port contention side channels in web browsers," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, (New York, NY, USA), p. 1182–1194, Association for Computing Machinery, 2022.

[17] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by key-extraction cache attacks from portable code," *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 119, 2018.

[18] G. Maisuradze and C. Rossow, "Ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, (New York, NY, USA), p. 2109–2122, Association for Computing Machinery, 2018.

[19] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "Thieves in the browser: Web-based cryptojacking in the wild," in *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19, Association for Computing Machinery, 2019.

[20] E. Tekiner, A. Acar, A. S. Uluagac, E. Kirda, and A. A. Selcuk, "In-browser cryptomining for good: An untold story," in *2021 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, pp. 20–29, 2021.

[21] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna, "Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1714–1730, 2018.

[22] A. Romano, Y. Zheng, and W. Wang, "Minerray: Semantics-aware analysis for ever-evolving cryptojacking detection," in *Proceedings of the 35th*

*IEEE/ACM International Conference on Automated Software Engineering*, pp. 1129–1140, 2020.

[23] F. N. Naseem, A. Aris, L. Babun, E. Tekiner, and A. S. Uluagac, "Minos: A lightweight real-time cryptojacking detection system.," in *NDSS*, 2021.

[24] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao, "Seismic: Secure in-lined script monitors for interrupting cryptojacks," in *Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II 23*, pp. 122–142, Springer, 2018.

[25] J. D. P. Rodriguez and J. Posegga, "Rapid: Resource and api-based detection against in-browser miners," in *Proceedings of the 34th Annual Computer Security Applications Conference*, pp. 313–326, 2018.

[26] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, N. Borisov, M. Antonakakis, and M. Bailey, "Outguard: Detecting in-browser covert cryptocurrency mining in the wild," in *The World Wide Web Conference*, pp. 840–852, 2019.

[27] H. Okhravi, M. Rabe, T. Mayberry, W. Leonard, T. Hobson, D. Bigelow, and W. Streilein, "Survey of cyber moving targets," *Massachusetts Inst of Technology Lexington Lincoln Lab, No. MIT/LL-TR-1166*, 2013.

[28] F. B. Cohen, "Operating system protection through program evolution.," *Computers & Security*, vol. 12, no. 6, pp. 565–584, 1993.

[29] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No.97TB100133)*, pp. 67–72, 1997.

[30] M. Eichin and J. Rochlis, "With microscope and tweezers: an analysis of the internet virus of november 1988," in *Proceedings. 1989 IEEE Symposium on Security and Privacy*, pp. 326–343, 1989.

[31] J. Cabrera Arteaga, "Artificial software diversification for webassembly," 2022. QC 20220909.

[32] A. Rossberg, B. L. Titzer, A. Haas, D. L. Schuff, D. Gohman, L. Wagner, A. Zakai, J. F. Bastien, and M. Holman, "Bringing the web up to speed with webassembly," *Commun. ACM*, vol. 61, p. 107–115, nov 2018.

[33] D. Bryant, "Webassembly outside the browser: A new foundation for pervasive computing," in *Proc. of ICWE 2020*, pp. 9–12, 2020.

[34] B. Spies and M. Mock, "An evaluation of webassembly in non-web environments," in *2021 XLVII Latin American Computing Conference (CLEI)*, pp. 1–10, 2021.

[35] E. Wen and G. Weber, "Wasmachine: Bring iot up to speed with a webassembly os," in *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pp. 1–4, IEEE, 2020.

[36] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world webassembly binaries: Security, languages, use cases," *Proceedings of the Web Conference 2021*, 2021.

[37] L. Wagner, M. Mayer, A. Marino, A. Soldani Nezhad, H. Zwaan, and I. Malavolta, "On the energy consumption and performance of webassembly binaries across programming languages and runtimes in iot," in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, EASE '23, (New York, NY, USA), p. 72–82, Association for Computing Machinery, 2023.

[38] N. Mäkitalo, T. Mikkonen, C. Pautasso, V. Bankowski, P. Daubaris, R. Mikkola, and O. Beletski, "Webassembly modules as lightweight containers for liquid iot applications," in *International Conference on Web Engineering*, pp. 328–336, Springer, 2021.

[39] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: A serverless-first, light-weight wasm runtime for the edge," in *Proceedings of the 21st International Middleware Conference*, p. 265–279, 2020.

[40] R. Gurdeep Singh and C. Scholliers, "Warduino: A dynamic webassembly virtual machine for programming microcontrollers," in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2019, (New York, NY, USA), pp. 27–36, ACM, 2019.

[41] I. Bastys, M. Algehed, A. Sjösten, and A. Sabelfeld, "Secwasm: Information flow control for webassembly," in *Static Analysis* (G. Singh and C. Urban, eds.), (Cham), pp. 74–103, Springer Nature Switzerland, 2022.

[42] T. Brito, P. Lopes, N. Santos, and J. F. Santos, "Wasmati: An efficient static vulnerability scanner for webassembly," *Computers & Security*, vol. 118, p. 102745, 2022.

[43] F. Marques, J. Fragoso Santos, N. Santos, and P. Adão, "Concolic execution for webassembly (artifact)," Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[44] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, ", : Sfi safety for native-compiled wasm," *Network and Distributed Systems Security (NDSS) Symposium*.

[45] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "Ct-wasm: Type-driven secure cryptography for the web ecosystem," *Proc. ACM Program. Lang.*, vol. 3, jan 2019.

[46] R. M. Tsoupidi, M. Balliu, and B. Baudry, "Vivienne: Relational verification of cryptographic implementations in webassembly," in *2021 IEEE Secure Development Conference (SecDev)*, pp. 94–102, 2021.

[47] Q. Stiévenart and C. De Roover, "Wassail: a webassembly static analysis library," in *Fifth International Workshop on Programming Technology for the Future Web*, 2021.

[48] F. Breitfelder, T. Roth, L. Baumgärtner, and M. Mezini, "Wasma: A static webassembly analysis framework for everyone," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 753–757, 2023.

[49] W. Fu, R. Lin, and D. Inge, "Taintassembly: Taint-based information flow control tracking for webassembly," *arXiv preprint arXiv:1802.01050*, 2018.

[50] D. Lehmann, M. T. Torp, and M. Pradel, "Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly," *arXiv preprint arXiv:2110.15433*, 2021.

[51] Q. Stiévenart, D. Binkley, and C. De Roover, "Dynamic slicing of webassembly binaries," in *39th IEEE International Conference on Software Maintenance and Evolution*, IEEE, 2023.

[52] Q. Stiévenart, D. W. Binkley, and C. De Roover, "Static stack-preserving intra-procedural slicing of webassembly binaries," in *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, (New York, NY, USA), p. 2031–2042, Association for Computing Machinery, 2022.

[53] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1045–1058, 2019.

[54] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, "Swivel: Hardening WebAssembly against spectre," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1433–1450, USENIX Association, Aug. 2021.

[55] M. Kolosick, S. Narayan, E. Johnson, C. Watt, M. LeMay, D. Garg, R. Jhala, and D. Stefan, "Isolation without taxation: Near-zero-cost transitions for webassembly and sfi," *Proc. ACM Program. Lang.*, vol. 6, jan 2022.

[56] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown, "Wave: a verifiably secure webassembly sandboxing runtime," in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2940–2955, 2023.

[57] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "New kid on the web: A study on the prevalence of webassembly in the wild," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*, pp. 23–42, Springer, 2019.

[58] S. Bhansali, A. Aris, A. Acar, H. Oz, and A. S. Uluagac, "A first look at code obfuscation for webassembly," in *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '22, (New York, NY, USA), p. 140–145, Association for Computing Machinery, 2022.

[59] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *ACM Comput. Surv.*, vol. 48, sep 2015.

[60] K. Pohl, G. Böckle, and F. Van Der Linden, *Software product line engineering: foundations, principles, and techniques*, vol. 1. Springer, 2005.

[61] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, (New York, NY, USA), p. 124–134, Association for Computing Machinery, 2011.

[62] Avizienis and Kelly, "Fault tolerance by design diversity: Concepts and experiments," *Computer*, vol. 17, no. 8, pp. 67–80, 1984.

[63] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive random testing: The art of test case diversity," *J. Syst. Softw.*, vol. 83, pp. 60–66, 2010.

[64] T. Jackson, *On the Design, Implications, and Effects of Implementing Software Diversity for Security.* PhD thesis, University of California, Irvine, 2012.

[65] G. R. Lundquist, V. Mohan, and K. W. Hamlen, "Searching for software0 diversity: Attaining artificial diversity through program synthesis," in *Proceedings of the 2016 New Security Paradigms Workshop*, NSPW '16, (New York, NY, USA), p. 80–91, Association for Computing Machinery, 2016.

[66] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Trans. Softw. Eng.*, vol. 12, p. 96–109, jan 1986.

[67] B. Randell, "System structure for software fault tolerance," *SIGPLAN Not.*, vol. 10, p. 437–449, apr 1975.

[68] N. Harrand, *Software Diversity for Third-Party Dependencies*. PhD thesis, KTH, Software and Computer systems, SCS, 2022. QCR 20220413.

[69] J. V. Cleemput, B. Coppens, and B. De Sutter, "Compiler mitigations for time attacks on modern x86 processors," *ACM Trans. Archit. Code Optim.*, vol. 8, jan 2012.

[70] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 1–11, IEEE, 2013.

[71] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a board range of memory error exploits," in *Proceedings of the USENIX Security Symposium*, 2003.

[72] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *Proceedings of the USENIX Security Symposium*, pp. 271–286, 2005.

[73] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, (New York, NY, USA), p. 16–27, Association for Computing Machinery, 1990.

[74] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity.," in *NDSS*, pp. 8–11, 2015.

[75] A. Romano, D. Lehmann, M. Pradel, and W. Wang, "Wobfuscator: Obfuscating javascript malware via opportunistic translation to webassembly," in *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, (Los Alamitos, CA, USA), pp. 1101–1116, IEEE Computer Society, may 2022.

[76] M. T. Aga and T. Austin, "Smokestack: thwarting dop attacks with runtime stack layout randomization," in *Proc. of CGO*, pp. 26–36, 2019.

[77] S. Lee, H. Kang, J. Jang, and B. B. Kang, "Savior: Thwarting stack-based memory safety violations by randomizing stack layout," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[78] Y. Younan, D. Pozza, F. Piessens, and W. Joosen, "Extended protection against stack smashing attacks without performance loss," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pp. 429–438, 2006.

[79] Y. Xu, Y. Solihin, and X. Shen, "Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization," in *Proc. of ASPLOS*, pp. 987–1000, 2020.

[80] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proc. of CCS*, pp. 272–280, 2003.

[81] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proc. CCS*, pp. 281–289, 2003.

[82] M. Chew and D. Song, "Mitigating buffer overflows by operating system randomization," Tech. Rep. CS-02-197, Carnegie Mellon University, 2002.

[83] D. Couroussé, T. Barry, B. Robisson, P. Jaillon, O. Potin, and J.-L. Lanet, "Runtime code polymorphism as a protection against side channel attacks," in *IFIP International Conference on Information Security Theory and Practice*, pp. 136–152, Springer, 2016.

[84] S. Cao, N. He, Y. Guo, and H. Wang, "WASMixer: Binary Obfuscation for WebAssembly," *arXiv e-prints*, p. arXiv:2308.03123, Aug. 2023.

[85] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. N. Saw, and R. Venkatesan, "The superdiversifier: Peephole individualization for software protection," in *International Workshop on Security*, pp. 100–120, Springer, 2008.

[86] M. Henry, "Superoptimizer: a look at the smallest program," *ACM SIGARCH Computer Architecture News*, vol. 15, pp. 122–126, Nov 1987.

[87] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, p. 216–226, 2014.

[88] B. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic program alignment for equivalence checking," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, (New York, NY, USA), p. 1027–1040, Association for Computing Machinery, 2019.

[89] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, p. 216–226, 2014.

[90] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, "Java decompiler diversity and its application to meta-decompilation," *Journal of Systems and Software*, vol. 168, p. 110645, 2020.

[91] M. Zalewski, "American fuzzy lop," 2017.

[92] K. Zhang, D. Wang, J. Xia, W. Y. Wang, and L. Li, "ALGO: Synthesizing Algorithmic Programs with Generated Oracle Verifiers," *arXiv e-prints*, p. arXiv:2305.14591, May 2023.

[93] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (C. R. Ramakrishnan and J. Rehof, eds.), (Berlin, Heidelberg), pp. 337–340, Springer Berlin Heidelberg, 2008.

[94] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati, "Scaling up superoptimization," *SIGARCH Comput. Archit. News*, vol. 44, p. 297–310, mar 2016.

[95] R. El-Khalil and A. D. Keromytis, "Hydan: Hiding information in program binaries," in *Information and Communications Security* (J. Lopez, S. Qing, and E. Okamoto, eds.), (Berlin, Heidelberg), pp. 187–199, Springer Berlin Heidelberg, 2004.

[96] V. Singhal, A. A. Pillai, C. Saumya, M. Kulkarni, and A. Machiry, "Cornucopia: A framework for feedback guided generation of binaries," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, (New York, NY, USA), Association for Computing Machinery, 2023.

[97] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: a secretless framework for security through diversity," in *Proc. of USENIX Security Symposium*, USENIX-SS'06, 2006.

[98] D. Bruschi, L. Cavallaro, and A. Lanzi, "Diversified process replicæ for defeating memory error exploits," in *Proc. of the Int. Performance, Computing, and Communications Conference*, 2007.

[99] B. Salamat, A. Gal, T. Jackson, K. Manivannan, G. Wagner, and M. Franz, "Stopping buffer overflow attacks at run-time: Simultaneous multi-variant program execution on a multicore processor," tech. rep., Technical Report 07-13, School of Information and Computer Sciences, UCIrvine, 2007.

[100] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *NDSS*, 2015.

[101] G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale, "The MEET approach: Securing cryptographic embedded software against side channel attacks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 8, pp. 1320–1333, 2015.

[102] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, "Compiler-generated software diversity," in *Moving Target Defense*, pp. 77–98, Springer, 2011.

[103] A. Amarilli, S. Müller, D. Naccache, D. Page, P. Rauzy, and M. Tunstall, "Can code polymorphism limit information leakage?," in *IFIP International Workshop on Information Security Theory and Practices*, pp. 1–21, Springer, 2011.

[104] A. Voulimeneas, D. Song, P. Larsen, M. Franz, and S. Volckaert, "dmvx: Secure and efficient multi-variant execution in a distributed setting," in *Proceedings of the 14th European Workshop on Systems Security*, pp. 41–47, 2021.

[105] R. Tsoupidi, R. C. Lozano, and B. Baudry, "Constraint-based diversification of JOP gadgets," *CoRR*, vol. abs/2111.09934, 2021.

[106] J. Cabrera Arteaga, O. Floros, O. Vera Perez, B. Baudry, and M. Monperrus, "Crow: code diversification for webassembly," in *MADWeb, NDSS 2021*, 2021.

[107] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the International Conference on Automated Software Engineering*, pp. 313–324, 2014.

[108] H. Bostani and V. Moonsamy, "Evadedroid: A practical evasion attack on machine learning for black-box android malware detection," *CoRR*, vol. abs/2110.03301, 2021.

[109] D. Yao, X. Shu, L. Cheng, S. J. Stolfo, E. Bertino, and R. Sandhu, *Anomaly detection as a service: challenges, advances, and opportunities*. Springer, 2018.

[110] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *J. Comput. Secur.*, vol. 6, p. 151–180, aug 1998.

[111] J. Cabrera Arteaga, M. Monperrus, and B. Baudry, "Scalable comparison of javascript v8 bytecode traces," in *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2019, (New York, NY, USA), p. 22–31, Association for Computing Machinery, 2019.

[112] Y. Fang, C. Huang, L. Liu, and M. Xue, "Research on malicious javascript detection technology based on lstm," *IEEE Access*, vol. 6, pp. 59118–59125, 2018.

[113] C. Fred, "Computer viruses," in *Proceedings of the 7th DoD/NBS Computer Security Conference 1984*, pp. 240–263, 1986.

[114] R. L. Castro, C. Schmitt, and G. D. Rodosek, "Armed: How automatic malware modifications can evade static detection?," in *2019 5th International Conference on Information Management (ICIM)*, pp. 20–27, 2019.

[115] R. L. Castro, C. Schmitt, and G. Dreo, "Aimed: Evolving malware with genetic programming to evade detection," in *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pp. 240–247, IEEE, 2019.

[116] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel, "When malware is packin' heat; limits of machine learning classifiers based on static analysis features," in *Proc. of NDSS*, 2020.

[117] M. Chua and V. Balachandran, "Effectiveness of android obfuscation on evading anti-malware," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, Association for Computing Machinery, 2018.

[118] P. Dasgupta and Z. Osman, "A Comparison of State-of-the-Art Techniques for Generating Adversarial Malware Binaries," *arXiv e-prints*, Nov. 2021.

[119] G. Lu and S. K. Debray, "Weaknesses in defenses against web-borne malware - (short paper)," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 10th International Conference, DIMVA. Proceedings* (K. Rieck, P. Stewin, and J. Seifert, eds.), Lecture Notes in Computer Science, 2013.

[120] M. Payer, "Embracing the new threat: Towards automatically self-diversifying malware,"

[121] N. Loose, F. Mächtle, C. Pott, V. Bezsmertnyi, and T. Eisenbarth, "Madvex: Instrumentation-based Adversarial Attacks on Machine Learning Malware Detection," *arXiv e-prints*, p. arXiv:2305.02559, May 2023.

[122] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, "Souper: A Synthesizing Superoptimizer," *arXiv preprint 1711.04422*, 2017.

[123] J. Cabrera Arteaga, P. Laperdrix, M. Monperrus, and B. Baudry, "Multi-Variant Execution at the Edge," *arXiv e-prints*, p. arXiv:2108.08125, Aug. 2021.

[124] J. Lettner, D. Song, T. Park, P. Larsen, S. Volckaert, and M. Franz, "Partisan: fast and flexible sanitization via run-time partitioning," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 403–422, Springer, 2018.

[125] B. G. Ryder, "Constructing the call graph of a program," *IEEE Transactions on Software Engineering*, no. 3, pp. 216–226, 1979.

[126] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, *et al.*, "Swivel: Hardening webassembly against spectre," in *USENIX Security Symposium*, 2021.

[127] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Sfi safety for native-compiled wasm," *NDSS. Internet Society*, 2021.

[128] J. Cabrera-Arteaga, N. Fitzgerald, M. Monperrus, and B. Baudry, "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly," *arXiv e-prints*, p. arXiv:2309.07638, Sept. 2023.

[129] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," *Proc. ACM Program. Lang.*, vol. 5, jan 2021.

[130] "Stop a wasm compiler bug before it becomes a problem | fastly." `https://www.fastly.com/blog/defense-in-depth-stopping-a-wasm-compiler-bug-before-it-became-a-problem`, 2021.

[131] D. Cao, R. Kunkel, C. Nandi, M. Willsey, Z. Tatlock, and N. Polikarpova, "Babble: Learning better abstractions with e-graphs and anti-unification," *Proc. ACM Program. Lang.*, vol. 7, jan 2023.

[132] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, (New York, NY, USA), p. 264–276, Association for Computing Machinery, 2009.

[133] T. D. Morgan and J. W. Morgan, "Web timing attacks made practical," *Black Hat*, 2015.

[134] T. Schnitzler, K. Kohls, E. Bitsikas, and C. Pöpper, "Hope of delivery: Extracting user locations from mobile instant messengers," in *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*, The Internet Society, 2023.

[135] Mozilla, "Protections Against Fingerprinting and Cryptocurrency Mining Available in Firefox Nightly and Beta ," 2019.

[136] J. Cabrera-Arteaga, M. Monperrus, T. Toady, and B. Baudry, "Webassembly diversification for malware evasion," *Computers & Security*, vol. 131, p. 103296, 2023.

[137] F. Cohen, "Computer viruses: theory and experiments," *Computers & security*, vol. 6, no. 1, pp. 22–35, 1987.

[138] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1–19, 2019.

[139] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript," in *Financial Cryptography and Data Security* (A. Kiayias, ed.), (Cham), pp. 247–267, Springer International Publishing, 2017.

[140] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, (New York, NY, USA), p. 151–163, Association for Computing Machinery, 2020.

# Part II

# Included papers

# SUPEROPTIMIZATION OF WEBASSEMBLY BYTECODE

**Javier Cabrera-Arteaga**, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
*Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs*

https://doi.org/10.1145/3397537.3397567

# CROW: CODE DIVERSIFICATION FOR WEBASSEMBLY

**Javier Cabrera-Arteaga**, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
*Network and Distributed System Security Symposium (NDSS 2021), MADWeb*

https://doi.org/10.14722/madweb.2021.23004

# MULTI-VARIANT EXECUTION AT THE EDGE

**Javier Cabrera-Arteaga**, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
*Conference on Computer and Communications Security (CCS 2022), Moving Target Defense (MTD)*

https://dl.acm.org/doi/abs/10.1145/3560828.3564007

# WEBASSEMBLY DIVERSIFICATION FOR MALWARE EVASION

**Javier Cabrera-Arteaga**, Tim Toady, Martin Monperrus, Benoit Baudry
*Computers & Security, Volume 131, 2023*

https://www.sciencedirect.com/science/article/pii/S0167404823002067

# WASM-MUTATE: FAST AND EFFECTIVE BINARY DIVERSIFICATION FOR WEBASSEMBLY

**Javier Cabrera-Arteaga**, Nick Fitzgerald, Martin Monperrus, Benoit Baudry
*Under revision*

# SCALABLE COMPARISON OF JAVASCRIPT V8 BYTECODE TRACES

**Javier Cabrera-Arteaga**, Martin Monperrus, Benoit Baudry
*11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019)*

https://doi.org/10.1145/3358504.3361228